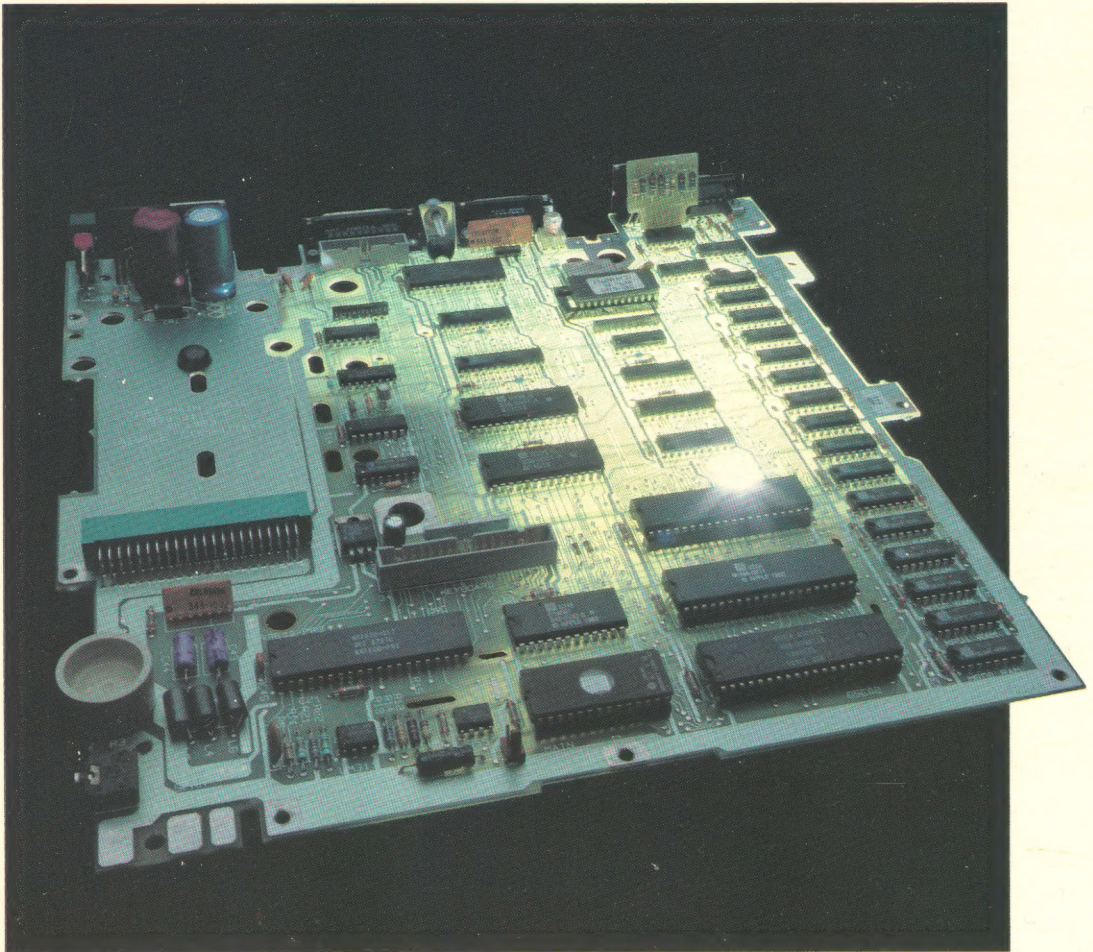


INSIDE THE APPLE IIc



Brady

Gary B. Little

Inside the *Apple //c*

Gary B. Little

Brady Communications Company, Inc.
A Simon & Schuster Publishing Company
New York, NY 10020

Inside the Apple IIc.

Copyright © 1985 by Brady Communications Company, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Brady Communications Company, Inc., A Simon & Schuster Publishing Company, 1230 Avenue of the Americas, New York, NY 10020.

Library of Congress Cataloging in Publication Data

Little, Gary B., 1954–
Inside the Apple IIc.

On t.p. IIc appears as //c.

Includes bibliographies and index.

1. Apple IIc (Computer) I. Title: Inside the Apple
2c. II. Title: Inside the Apple two c. III. Title.

QA76.8.A66225L57 1985 001.64 84-27419

ISBN 0-89303-564-5

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95

2 3 4 5 6 7 8 9 10

Production Editor/Text Designer: Michael J. Rogers

Art Director: Don Sellers

Assistant Art Director: Bernard Vervin

Cover Photography: George Dodson

Manufacturing Director: John A. Komsa

Copy Editor: Rita Proglor

Typesetting: Automated Graphic Systems, White Plains, MD

Printing: R. R. Donnelley & Sons Co., Harrisonburg, VA

Typefaces: Helvetica (display), Aster (text), and Universal Monotype #3 H-P (computer programs)

To my wife, Pamela

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Note to Authors

Have you written a book related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. Brady produces a complete range of books for the personal computer market. We invite you to write to Editorial Dept., Brady Communications Co., A Simon & Schuster Publishing Company, 1230 Avenue of the Americas, New York, NY 10020.

Trademarks of Material Mentioned in This Text

Apple //e, Applesoft, Apple II, Apple II Plus, Apple //c, Apple I, Integer BASIC, DOS 3.3, Lisa, Macintosh, and ProDOS are trademarks of Apple Computer, Inc.

Contents

Preface / xlii

1 An Introduction to Apple and the Apple //c / 1

A Condensed History of Apple Computer, Inc. / 1

Hardware and the Apple //c / 7

Learning the Fundamentals / 8

What Won't Be Covered / 9

Using the Optional Diskette / 9

Further Reading for Chapter 1 / 10

2 The 65C02 Microprocessor / 11

Important 65C02 Concepts / 12

Zero Page and the Stack / 12

65C02 Instruction Set / 13

65C02 Registers / 21

The Accumulator—A / 21

The Index Registers—X and Y / 22

The Processor Status Register—P / 23

Carry Flag (C) / 23

Zero Flag (Z) / 24

Interrupt Disable Flag (I) / 24

Decimal Mode Flag (D) / 25

Break Flag (B) / 25

Overflow Flag (V) / 25

Negative Flag (N) / 26

The Stack Pointer—S / 26

The Program Counter—PC / 27

65C02 Addressing Modes / 27

Immediate / 28

Absolute / 29

Accumulator / 30

Implied / 30

Zero-Page Indexed Indirect / 30

Zero-Page Indirect / 31

Indirect Indexed / 31

Absolute Indexed / 32

Relative / 32

Absolute Indirect / 33

Absolute Indexed Indirect / 33

65C02 Input/Output Handling / 33

65C02 Interrupts / 34

Reset Interrupt / 36

Interrupt Request (IRQ) /	36
The BRK Instruction /	37
The 65C02 Memory Space on the //c /	38
RAM Memory /	38
Input/Output (I/O) Memory /	41
ROM Memory /	42
Further Reading for Chapter 2 /	42
3 The System Monitor /	45
The System Monitor Commands /	46
The DISPLAY Command : Displaying the Contents of Memory /	46
The STORE Command : Changing the Contents of Memory /	49
The MOVE Command : Copying the Contents of Memory /	51
The VERIFY Command : Comparing Ranges of Memory /	53
The EXAMINE Command : Examining the 65C02's Registers /	53
The GO Command : Running a Program /	54
The LIST Command : Disassembling Assembly-Language Programs /	55
The NORMAL and INVERSE Commands: Changing Video Display Modes /	57
The ADD and SUBTRACT Commands : Simple Arithmetic /	57
The BASIC and CONTINUE BASIC Commands : Entering Applesoft /	57
The USER Command : User-Defined Command /	58
The KEYBOARD and PRINTER Commands : Redirecting Input and Output /	59
Multiple Commands on One Line /	61
System Monitor Subroutines /	61
Further Reading for Chapter 3 /	65
4 Applesoft BASIC /	67
Applesoft Memory Map /	68
Tokenization of Applesoft Programs /	73
Keyword Tokens /	74
Storage of Applesoft Variables /	77
Storage of Simple Variables /	78
The Name Header /	79
The Data Field /	80
End of Simple Variables /	82
Storage of Array Variables /	83
The Name Header /	83
Dimensioning Bytes /	83
The Data Field /	85
End of Array Variables /	85
Representation of Integer Numbers /	85
Representation of Real Numbers /	87

Number Theory /	87
Binary Floating-Point Format /	87
How an Applesoft Program Runs /	89
The CHARGET Subroutine /	91
Changing Program Flow /	93
Finding Line Numbers /	93
Linking Applesoft to Assembly-Language Programs /	94
The CALL Command /	95
The & Command /	95
The USR Function /	96
Applesoft's Built-In Subroutines /	97
Using Applesoft's Built-In Subroutines /	104
Locating Variables /	104
Evaluating Formulas /	108
Converting Numbers /	108
Further Reading for Chapter 4 /	112
5 The ProDOS Disk Operating System /	115
Formatting Diskettes /	116
ProDOS Memory Map /	116
ProDOS Page 3 Vectors /	118
Filenames and Pathnames /	118
BASIC.SYSTEM Commands /	121
File Management Commands /	122
File Loading and Execution Commands /	123
File Input/Output Commands /	124
Miscellaneous Commands /	125
ProDOS File Storage /	126
Volume Bit Map /	126
Diskette Directory /	127
"Protecting" Files /	130
Storing File Data /	131
MLI—Accessing the Diskette Directly /	132
READ.BLOCK Program /	134
Further Reading for Chapter 5 /	140
6 Character Input and the Keyboard /	141
Standard Character Input Subroutines /	145
Reading One Character /	146
RDKEY (\$FD0C) /	146
Keyboard Input /	148
Escape Sequences /	148
RDCHAR (\$FD35) and ESCRDKEY (\$CCED) /	150
Reading a Line of Characters /	150
Changing Input Devices : The Input Link /	152
How About Output? /	153

Designing a KSW Input Subroutine /	153
Replacing the Keyboard Input Subroutine /	153
ProDOS and the Input Link /	154
The Keyboard /	157
Encoding of Keyboard Characters /	157
Special Keys /	159
The "Apple" Keys /	159
Keyboard I/O Locations /	160
Modifying the Keyboard Input Subroutine /	164
Keyboard Auto-Repeat /	168
Resetting the Apple //c /	173
Special RESET Procedure /	173
Trapping "Soft" RESETs /	174
Trapping RESET from Assembly Language /	175
Trapping RESET from Applesoft /	176
Further Reading for Chapter 6 /	181
7 Character and Graphic Output and Video Display Modes /	183
Text Mode /	184
The 80/40 Switch /	185
Turning on the Text Display /	185
Text Mode Memory Mapping /	188
40-Column Text Mode /	189
80-Column Text Mode /	191
Using Page2 of Text /	192
Video Display Attributes: Normal, Inverse, Flash /	194
MouseText /	196
Standard Character Output Subroutines /	199
Video Output /	200
Video Screen Windowing /	202
How COUT1 and C3COUT1 Set the Video Attribute /	203
Changing Output Devices : The OUTPUT Link /	205
Designing a CSW Output Subroutine /	206
Replacing the Video Output Subroutine /	206
ProDOS and the Output Link /	206
Low-Resolution Graphics Mode /	207
Turning on the Low-Resolution Graphics Display /	208
Low-Resolution Graphics Screen Memory Mapping /	209
Low-Resolution Graphics Colors /	209
Double-Width Low-Resolution Graphics /	210
Turning on Double-Width Low-Resolution Graphics /	210
Double-Width Low-Resolution Graphics Screen Memory Mapping /	212
Double-Width Low-Resolution Graphics Colors /	212
Built-In Support for Low-Resolution Graphics /	213
High-Resolution Graphics Mode /	214

Turning on the High-Resolution Graphics Display /	215
High-Resolution Graphics Screen Memory Mapping /	217
High-Resolution Graphics Colors /	219
Animation with High-Resolution Graphics /	220
Double-Width High-Resolution Graphics /	221
Turning on Double-Width High-Resolution Graphics /	222
Double-Width High-Resolution Graphics Screen Memory Mapping /	222
Double-Width High-Resolution Graphics Colors /	223
Built-In Support for High-Resolution Graphics /	223
Further Reading for Chapter 7 /	225
8 Memory Management /	229
16K Bank-Switched RAM Areas /	230
Using Bank-Switched RAM /	231
Reading the Status of Bank-Switched RAM Soft Switches /	232
Auxiliary Bank-Switched RAM /	234
Playing with Bank-Switched RAM /	235
Bank-Switched RAM and ProDOS /	236
Auxiliary RAM Memory Area /	236
Using Auxiliary Memory /	237
The ALTZP Switch /	237
The RAMRD and RAMWRT Switches /	239
Auxiliary Memory Support Subroutines /	241
AUXMOVE (\$C311)—Transferring data to and from auxiliary memory /	241
XFER (\$C314)—Transferring control to a program from main or auxiliary memory /	244
Running Co-Resident Programs /	245
Initialization of the Auxiliary Stack /	251
Using CONCURRENT /	251
Limitations of CONCURRENT /	252
Further Reading for Chapter 8 /	253
9 The Speaker /	255
Generating Musical Notes /	255
Generating Music /	259
Further Reading for Chapter 9 /	263
10 Mouse and Game Controller Input /	265
The Apple Mouse /	265
How the Mouse Works /	266
Mouse Operating Modes /	267
Passive (Transparent) Mode /	267
Movement Interrupt Mode /	268
Button Interrupt Mode /	268

Movement or Button Interrupt Mode /	268
Vertical Blanking Interrupts /	268
The Mouse and Applesoft /	269
Turning the Mouse On /	269
Turning the Mouse Off /	271
Reading the Mouse /	271
A Sample Program /	272
The Mouse and Assembly Language /	273
Mouse Screen Hole Locations /	274
Using the Mouse Subroutines /	275
Comparing the //c Mouse with the //e Mouse /	275
The Mouse Subroutines /	278
A Sample Program /	280
The Mouse as a Joystick /	280
Mouse I/O Locations /	285
The Game Controller Interface /	288
Game Controller Inputs /	289
Push Button Inputs /	293
Further Reading for Chapter 10 /	296

11 The Serial Interface Ports / 299

Serial Transmission of Data /	299
The RS-232-C Standard /	300
Data Communications Protocols for Serial Communications /	300
Start Bit /	301
Data Bits /	301
Parity Bit /	302
Stop Bits /	302
Data Transmission Errors /	303
The 6551 ACIA /	303
6551 Control Register /	306
6551 Command Register /	307
6551 Status Register /	308
6551 Data Register /	310
Configuring the Serial Ports /	310
Characteristics of a Printer Port /	311
Characteristics of a Communications Port /	313
Terminal Mode /	314
Changing the Default Configuration /	316
6551 Interrupt Handling /	318
6551 Transmitter Interrupts /	319
6551 Receiver Interrupts /	319
6551 Keyboard (DSR port 2) Interrupts /	321
6551 External (DSR port 1) Interrupts /	323
Further Reading for Chapter 11 /	324

Appendix I / 325

American National Standard Code for Information Interchange (ASCII)
Character Codes

Appendix II / 331

65C02 Instruction Set and Cycle Times

Appendix III / 337

Apple //c Soft Switch, Status, and I/O Port Locations / **337**
I/O Port Locations / **343**

Appendix IV / 347

Apple //c Page 3 Vectors

Appendix V For Beginners Only / 349

Numbering Systems / **349**
Bit Numbering and "Significance" / **350**
Pointers and Vectors / **351**
Control Characters / **351**
65C02 Assembly Language / **351**
Running Assembly-Language Programs / **353**

Appendix VI Periodicals of Interest / 355**Index / 357**

Preface

If you bought your Apple //c in order to do all sorts of strange things to it: POKE around in it, PEEK inside it, CALL subroutines, RUN programs, and so on . . . If you thrill in making a computer do things that its designers never imagined . . . If you write and debug programs for fun . . . If you have an Apple bumper sticker on your car . . . This book is for you!

In this book, we're going to explore all the important software nooks and crannies in the //c and see how to exploit the power that they hold. You will be expected to be proficient in the Applesoft BASIC language; an understanding of 65C02 assembly language will also be invaluable.

Some of the major topics that will be covered are as follows:

- The 65C02 microprocessor that controls the //c. This will include a discussion of 65C02 instructions, addressing modes, I/O handling, and interrupt handling.
- The //c system monitor commands, the structure of the Applesoft language, and the internal structure of ProDOS.
- How the //c handles character input and output. This includes a discussion of keyboard input and the various video display modes supported by the //c (text, graphics, and double-width graphics).
- Memory management techniques.
- How to control the speaker, mouse, and game controller.
- How to use the //c's two built-in serial ports for communication with printers and modems.

After you have read this book, you will know absolutely everything there is to know about how the //c interacts with the outside world and how it processes information. (Well, almost everything.) Descriptions of all the "soft switches" that the //c uses to control its hardware environment will be presented, as will examples of how to use the //c's I/O memory locations. Furthermore, many of the more important subroutines contained in the //c's ROM area will be analyzed and explained.

Here are some of the more interesting programming examples that will be presented in this book:

- How to speed up the auto-repeat rate of the cursor (using software techniques only).
- How to run two Applesoft programs concurrently (one in main memory and the other in auxiliary memory).
- How to read mouse input using 65C02 interrupt techniques.
- How to read and write specific blocks on a ProDOS-formatted diskette.
- How to use the keyboard "type-ahead" feature.

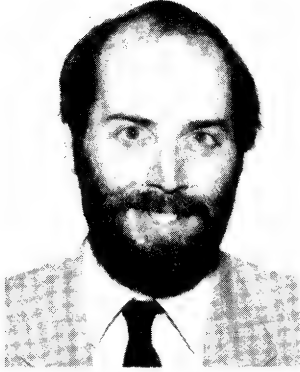
Complete and commented source listings for these programs and several others are included in the text. They are also available in machine-readable form on an optional diskette.

I hope that you enjoy reading this book as much as I enjoyed writing it. You'll find it a useful reference and an invaluable source of inspiration for the development of your own software.

My thanks to Rich Williams and Apple Computer, Inc. in Cupertino for helping me to decode the meaning of some of the more obscure code in the //c's ROM. Rich should know—he wrote most of it.

Gary B. Little
Vancouver, British Columbia, Canada
March 1985

About the Author



Gary B. Little is an expert Apple II (and II Plus, //e, //c, ...) programmer who resides in Vancouver, British Columbia. He is a founding member of the Apples British Columbia Computer Society and of SAGE (Serious Apple Group, Eh!) and is also an active member of several business organizations that promote and assist software developers. Gary has written numerous articles for several computer publications and is the author of one other micro-computer book published by Brady Communications, *Inside the Apple //e*.

An Introduction to Apple and the Apple //c

The Apple //c is the newest member of Apple Computer Inc.'s highly popular Apple II family of computers. The other members of this family are the original Apple II (1977), the Apple II Plus (1979), and the Apple //e (1983).

In this book we will be taking an advanced "inside" look at the Apple //c itself. Keep in mind, however, that much of what will be said will also apply to its three predecessors because Apple has made a substantial effort to maintain a high degree of compatibility with other members of the Apple II family. We will be concentrating on the //c's built-in language and operating system (Applesoft and the system monitor) and the ProDOS disk operating system; other languages and operating systems will be mentioned only briefly.

A Condensed History of Apple Computer, Inc.

Before we begin our detailed examination of the Apple //c, let's take a brief look at the history of Apple, the company. This history will reveal how the original Apple II slowly evolved into the Apple //c in 1984 and will serve to explain much of the rationale behind the design of the //c.

1976

In the beginning, Apple was made up of just two individuals: Stephen Wozniak ("Woz") and Steven Jobs. Woz provided the hardware and software expertise and almost single-handedly designed the company's first two computers, the Apple I and the Apple II (he had the help of Rod Holt who designed the Apple II's power supply). A patent application was subsequently filed with respect to the Apple II on April 11, 1977, and U.S. patent #4,136,359 was eventually issued in early 1979. Jobs was largely responsible for marketing and raising financing, and it was he who came up with the "Apple" name (Jobs was apparently thinking of a job that he had recently had in an Oregon orchard). In the early going, both partners were still working for other com-

puter companies in California's Silicon Valley: Jobs with Atari and Woz with Hewlett-Packard. Fortunately for Apple, Hewlett-Packard was not interested in Woz's design for a personal computer and gave him a release so that he could deal with it as he saw fit.

The Apple I was designed to be sold to and used by hobbyists only; in all, only about 175 were sold. The Apple II, however, was designed with a much larger market in mind (although Woz claims he simply wanted to build a computer with which he could play Atari's "Breakout" game). That market quickly materialized as a result of the startling combination (for 1977) of excellent hardware, attractive packaging, and the availability of informative technical reference material.

Woz decided to use the MOS Technology 6502 microprocessor to control the Apple II. This decision was dictated not by the 6502's reliability, powerful instruction set, or any other design characteristic, but rather by its price. Whereas other microprocessors were selling for hundreds of dollars in 1976 and were difficult to find, the 6502 was readily available and it cost only about twenty dollars. The //c uses the newer 65C02 microprocessor, but it recognizes all the instructions that the original 6502 uses and supports a few new ones as well.

With assistance from Allen Baum, Woz wrote all the software for the original Apple II that was stored in its read-only memory (ROM). This included a version of the BASIC programming language called Integer BASIC (which can't handle decimal numbers but is great for games), a system monitor for debugging and for handling fundamental input/output operations, a set of mathematical subroutines, a mini-assembler for entering programs in assembly language, and "Sweet 16," a software-simulated 16-bit microprocessor (Woz was way ahead of his time).

To raise a little money for their fledgling venture, Woz sold his Hewlett-Packard pocket calculator and Jobs sold his Volkswagen bus. Overhead expenses were cut to the bare minimum by setting up operation in the garage of Jobs' parents. As 1977 rolled around, however, it became clear that more money, a lot more money, was going to be needed.

1977

Since Jobs was the partner responsible for marketing the Apple II, it was he who began searching for venture capital. That search eventually led him to Mike Markkula, a former marketing manager at Intel, an integrated-circuit designing company. Markkula, Jobs, and Wozniak quickly struck a deal whereby Markkula agreed to put a quarter of a million dollars into Apple in exchange for an equal partnership interest. He then proceeded to use his expertise to line up bank financing and additional capital funding. Apple was then finally ready for the mass market!

The Apple II was formally announced for sale at the first annual West Coast Computer Faire in early 1977 and it was an instant success. The main reasons for its early success were that it was easily expandable (more memory could easily be added to it and eight slots were available for peripheral devices when they became available), it had a full-size keyboard, and it had *color* graphics. And, yes, it looked great!

Not that there weren't any problems, however. For example, lower case characters could not be produced by the keyboard and the video display was only forty columns wide. These shortcomings officially persisted until the introduction of the Apple //e in 1983, although several other sources of upper- and lowercase keyboards and 80-column boards did pop up in the interim.

One software problem had to be remedied quickly. Integer BASIC did not support decimal (floating-point) numbers or functions, and so business and scientific use of the Apple II was necessarily limited. Apple began to take steps to remedy this in the summer of 1977 when it negotiated the purchase of about ten thousand lines of program source code for a floating-point version of BASIC from Microsoft Corporation. This code was written in 6502 assembly language and so could be readily adapted to run on the Apple II.

By this time Apple had a few employees, one of which was a young programmer by the name of Randy Wigginton. Wigginton reworked the Microsoft source code and came out with a preliminary version of a floating-point BASIC that would run on the Apple II. This version was called "Applesoft—Extended Precision Floating Point BASIC Language" and was released in October 1977. Further work was required to polish Applesoft into a final product, and this was done during the winter of 1977.

1978

The final version of Applesoft, Applesoft][, was finally released in May 1978; this same version, with some minor changes, is still in use today on the Apple //c. It was first available on cassette tape only, but was later provided in ROM on a card that could be plugged into a slot on the Apple II; it eventually replaced Integer BASIC on the motherboard when the Apple II Plus was released in 1979.

The most important new product released in 1978 was probably the Disk II disk drive and controller card peripherals that are now built in to the Apple //c. The disk drive revolutionized the software business because for the first time it was feasible to develop sophisticated programs that could be easily loaded and that could quickly and reliably access large data bases. Until the disk drive was released, all programs had to be saved to and loaded from cassette tape, which was invariably an exercise in frustration. Many a cottage software business started up after the disk drive became available, and in a short time hundreds of commercial software products were being developed for the Apple II.

The Disk II was controlled by a program called the Disk Operating System (DOS), first written by Bob Shepardson and later substantially modified by Randy Wigginton, J. R. Huston, and Rick Auricchio. DOS has undergone several revisions throughout the years and the current version is DOS 3.3. This version still works with the Apple //c, although it has been superseded by a new DOS called ProDOS.

1979

Sales really ballooned for Apple in 1979. Apple was able to increase sales by a total of forty million dollars (!) over the previous year, to a total of forty-eight million dollars. By this time, the Apple II was selling not only because it was an excellent hardware package but also because an ever-increasing supply of software was available that could be run on it. One important piece of software, VisiCalc, the very first financial spreadsheet program, is reputed to have been directly responsible for stimulating the purchase of tens of thousands of Apple II computers.

The Apple II underwent minor surgery in 1979 and came out of it with a new name: Apple II Plus. The Apple II Plus is essentially the same as an Apple II, except that its ROM chips contain Applesoft][rather than Integer BASIC and its system monitor supports more powerful screen-editing commands and the ability to automatically run a program from diskette whenever the power is turned on. At the same time, a couple of handy debugging commands (step and trace) were taken out of the system monitor, but they were not missed by many users. The modifications to the system monitor were written by John Arkley.

Apple announced its Pascal Operating System in 1979 as well. Because Pascal requires a huge amount of memory in which to operate, Apple also released a new peripheral card, called a language card, at the same time. The language card effectively added another 16K of memory to the Apple II, which could “replace” the Applesoft ROMs when Pascal was being used. The language card was plugged into slot #0 of the Apple II but in the //c it is simulated in the memory chips on the motherboard. These different implementations, however, are transparent to the user.

1980–1982

Apple’s sales continued to explode in the early eighties: \$117 million in 1980, \$334.8 million in 1981, and \$583.1 million in 1982! Most of these sales were generated by the Apple II Plus, which eventually set a record for monthly sales in December 1982.

The infamous Apple /// was released in 1980. For several reasons, notably its early unreliability and high price, it never established a significant market presence even though a modified version (known as the Apple /// Plus) was

still being produced as late as 1984. It comes with an Apple II emulation mode that allows it to run most, but not all, of the software that runs on the Apple II.

In the winter of 1980–81, Apple made a public offering of stock, which was quickly snapped up. The proceeds were largely directed into intensive (and expensive) research and development projects. We'll see in a moment what those projects led to.

If imitation is the sincerest form of flattery, then Apple must surely be blushing. Since about 1980, tens of thousands of unofficial Apple II "clones" (euphemistically called "compatibles") have been manufactured, mostly by Taiwanese concerns. To achieve absolute compatibility with the Apple II, most of these clones contain ROMs that are direct copies of the Applesoft and system monitor ROMs. Not surprisingly, Apple considers this to be highly improper and has successfully instituted legal proceedings in the United States and many other countries against several manufacturers in order to protect its copyrights and patent rights. The importation of Apple II clones to the United States has also been reduced because Apple has registered its copyrights with U.S. Customs. The Customs authorities have the power to confiscate shipments of products that violate Apple's copyrights.

1983

At Apple's Annual General Meeting on January 19, 1983, two major announcements were made. First, the Lisa computer was announced, a computer that was immediately recognized as a technological and innovative triumph because of its ease of use and excellent operating system. Its retail price, however, was initially too high for it to sell in the quantities that Apple would have liked. Subsequent price reductions, coupled with increasing availability of software, has helped to remedy this problem.

The second major announcement was the introduction of the successor to the Apple II Plus, the Apple //e. The Apple //e was carefully designed to maintain as high a degree of compatibility with the Apple II Plus as possible so that the thousands of software packages developed for the Apple II Plus would not have to be rewritten. Several new features were added to the //e, however, that make it a significantly different computer: built-in support for an 80-column display, an upper- and lowercase keyboard, self-testing subroutines, and enhanced editing capabilities.

In addition, Apple significantly simplified the construction of the //e by reducing the number of integrated circuits on the motherboard from 109 on the Apple II Plus to only 31! It did this by designing two special integrated circuits to replace many of the discrete components used on the II Plus.

The manager of the team that designed the Apple //e was Peter Quinn. The hardware was designed by Walt Broedner and most of the modifications to the old system monitor were made by Rick Auricchio and Bryan Stearns.



There was also a major change at the managerial level at Apple in 1983. On April 8, Apple announced that Mike Markkula had resigned as President and that John Sculley had been named to succeed him. Sculley was formerly president of Pepsi-Cola and it is reported that his salary is in excess of one million dollars per year.

1984

At its January 24, 1984, Annual General Meeting Apple announced the Macintosh computer ("Mac"), a scaled-down version of Lisa. Mac undoubtedly represents another mass-market best seller for Apple because it is easy to use and it is priced affordably. Within a month of its release, at least two Mac-specific magazines and several books had been published. This is reminiscent of what happened in 1979 when sales of the Apple II began to skyrocket.

For users of Apple II computers, there was one major announcement at the Annual General Meeting: the release of a successor to DOS 3.3 called ProDOS. This disk operating system is significantly different from, but upwardly compatible with, DOS 3.3. Most Applesoft programs, when transferred to ProDOS-formatted diskettes, will run without modification. The main advantages of ProDOS are that it is faster, it is easier for programmers to use, it supports a directory structure that is more convenient for use with larger-capacity diskettes or hard disks, and its disk format can be read by the Apple ///.

On April 24, 1984, Apple formally introduced the portable Apple //c. In keeping with Apple tradition, the //c is compatible with most software designed for its predecessor, the Apple //e. However, the //c actually represents a radical departure from the Apple II norm because of the way the hardware has been packaged. For example, the peripheral expansion slots on the //e are gone and have been replaced by built-in interfaces and a built-in disk drive. This was done primarily to reduce the size of the unit to that of a true portable, and also to make the //c appeal more to the large "plug 'n run" class of users. Furthermore, the //c simply looks different than any of the earlier Apple II computers.

The //c is a wonder of computer miniaturization. Not counting the sixteen RAM chips that provide the //c with 128K of memory, there are only twenty-one integrated circuits (ICs) used in the system. Several of these ICs are custom large-scale-integration (LSI) chips that perform tasks that are normally handled by several discrete ICs when conventional technology is used.

The manager of the //c design team was the same Peter Quinn who was in charge of the //e project. The //c's firmware was written by Ernie Beernink, Rich Williams, and James Huston; if you ever forget their names, press

[control-RESET] right after you turn on the //c (before the disk has a chance to boot up), and then type in and run the following short Applesoft program:

```
100 IN# 5: INPUT A$: PRINT A$
```

Surprise, surprise!

Hardware and the Apple //c

Although this book is primarily concerned with software, let's begin by taking a quick look at the hardware that makes up the Apple //c.

The keyboard is laid out in the standard QWERTY arrangement and contains all of the keys you would find on a standard typewriter plus a few extra special ones to boot. Just above the keyboard are the reset button, the 80/40 switch, and the keyboard switch. The keyboard and these special switches will be described in detail in Chapter 6.

On the left side of the //c you will find the volume control wheel for the speaker and the speaker headphone jack. You can't see the speaker because it's under the hood. In Chapter 9 we will see how it can be used to generate music.

On the other side of the //c is the built-in disk drive. This drive can read diskettes formatted on all previous Apple II models.

We won't encourage you to take the //c apart to see the electronic circuitry lurking beneath the surface since this is frowned on by the warranty people at Apple. If you did, however, you would see the various integrated circuits that make up the //c, including the 65C02 microprocessor that controls everything, the two 6551 serial communications adapters, and the RAM and ROM memory chips.

The //c can be interfaced to the "real world" through one of seven connectors that are located on its back panel (see Figure 1-1):

- The mouse/game connector: This is where the Apple Mouse or a joystick can be connected. We will be examining it in Chapter 10.
- The modem port (serial port 2): A modem is a device that allows you to communicate with other computers over standard telephone lines. See Chapter 11.
- The video expansion connector: This connector allows the //c to be connected to any of several display options, including RGB (red-green-blue) displays, flat-panel liquid-crystal displays, and ordinary black and white or color television sets.
- The video monitor connector.
- The external disk drive connector: For those who can't survive with just one drive.

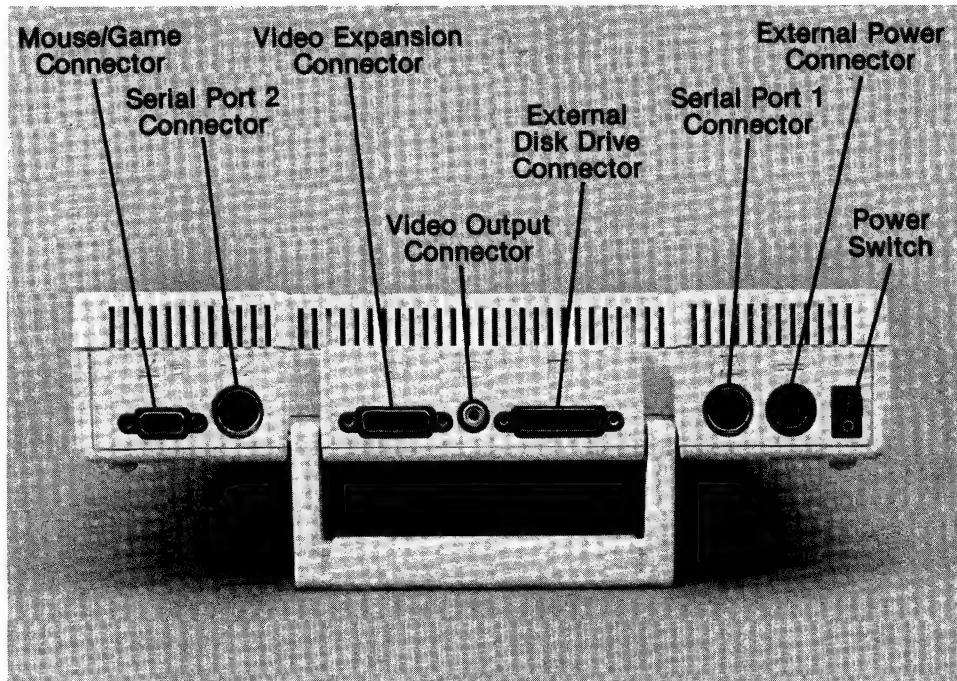


Figure 1-1. Back panel of the Apple //c.

- The printer port (serial port 1): A printer will undoubtedly be the first peripheral that you add to your //c. See Chapter 11.
- The power connector: You can't do much without it!

Previous members of the Apple II family have several expansion slots that can be used to hold interface cards that control external devices. The //c has no such slots, but its built-in interfaces and supporting software have been carefully designed so that they can be controlled by the same commands that would be used on a slot-based Apple II. The only difference is a semantical one: you refer to "port" numbers on the //c and to "slot" numbers on the Apple //e, Apple II Plus, and Apple II. The numbers assigned to each port on the //c are shown in Table 1-1.

So much for the //c's hardware!

Learning the Fundamentals

Most of the readers of this book are expected to be intermediate to advanced programmers who need no explanation of fundamental concepts such as numbering systems, bits and bytes, pointers, vectors, assembly language, and how to load and run programs.

Table 1-1. Port assignments on the Apple //c.

<i>Port Number</i>	<i>Description</i>
0	Standard keyboard and video I/O (see Chapters 6 and 7)
1	Serial interface for printer (see Chapter 11)
2	Serial interface for modem (see Chapter 11)
3	80-column video display (see Chapter 7)
4	Mouse interface (see Chapter 10)
6	Internal disk drive interface (see Chapter 5)
7	External disk drive interface (see Chapter 5)

If you feel a little uncomfortable with any of these concepts, then you should first read Appendix V ("For Beginners Only") before attempting to tackle the rest of this book. You will probably also feel more comfortable if you read an introductory book on computer systems first.

What Won't Be Covered

There are a few topics that will not be discussed at length in this book. Integer BASIC, the BASIC that was built into the first several thousand Apple II's, will not be discussed because it is rarely used anymore and is fast becoming obsolete. In fact, the ProDOS disk operating system does not allow Integer BASIC programs to be run at all.

The only high-level language that will be discussed at length will be Applesoft. For more information on Apple Pascal, Fortran, or Logo, you will have to consult other texts.

Using the Optional Diskette

This book can be purchased either with or without a ProDOS-formatted program diskette, or the diskette can be purchased separately. The diskette contains all the programs that are presented as examples in the following chapters and will allow you to quickly load a program into memory, or modify a program, without having to endure the pleasure of typing it in from scratch.

The files on the diskette are either Applesoft programs (marked by "BAS" when you CATALOG the diskette), text files (marked by "TXT"), or binary programs (marked by "BIN").

The text files on the diskette are the source-code listings for the binary programs and are in the format expected by the Merlin Pro assembler (which is available from Roger Wagner Publishing, Inc., 10761 Woodside Avenue,

Suite E, Santee, California). Most other assemblers are also able to read these text files. Keep in mind, however, that the source-code formats used by different assemblers do vary and it may be necessary to modify a source code file to take into account any such differences before the file can be properly assembled.

The Applesoft programs and binary programs can usually be run by using the standard RUN and BRUN commands, respectively, or the ProDOS “intelligent RUN command”, “-” (a dash), which automatically checks the file type and will RUN an Applesoft program or BRUN a binary program. Some of the binary programs, however, are designed to be called from an Applesoft program only and should simply be loaded into memory using the BLOAD command. Such exceptions will be noted in the discussions that relate to these programs in this book.

Further Reading for Chapter 1

Historical background . . .

- “Photograph of Apple I”, *Apple Orchard*, April 1983, front cover. The original Apple product.
- A.L. Taylor III, “Striking it Rich”, *Time*, February 15, 1982, pp. 42-47. Apple makes the front cover of *Time*.
- D. Garr, *Woz: The Prodigal Son of Silicon Valley*, Avon Books, 1984. An in-depth review of the history of Apple.
- P. Freiburger and M. Swaine, *Fire in the Valley: The Making of the Personal Computer*, Osborne/McGraw-Hill, 1984.
- M. Moritz, *The Little Kingdom: The Private Story of Apple Computer*, Morrow, 1984.
- P. Lopiccola, “Core of a New Apple”, *Popular Computing*, March 1983, pp. 114–117. How the Apple II Plus was transformed into the //c’s predecessor, the Apple //e.
- J. Markoff, “The Apple IIc Personal Computer”, *Byte*, May 1984, pp. 276–284. Refer to this article for pictures of the inside of the //c and a good overview of the //c’s hardware.

Standard reference work . . .

- The Apple //c Reference Manual, Volumes 1 and 2*, Apple Computer, Inc., 1984. Includes detailed information on the hardware and software that makes up the Apple //c. Source code for the //c system monitor is included.

2

The 65C02 Microprocessor

The “brains” of every microcomputer are represented by a complex integrated circuit called a microprocessor that controls the operation of the system as a whole. The microprocessor used in the //c is called a 65C02.

The 65C02 is closely related to the 6502 microprocessor that is used in the Apple //e, Apple II Plus, and Apple II. In fact, all programming instructions supported by the 6502 are also supported by the 65C02. This is fortunate since it means that no translation of specific 6502 instructions need be performed before the program can be executed by the 65C02 microprocessor. (This does not mean, of course, that any program written for the 6502-based Apple systems will run on the //c. If the program accesses subroutines or I/O locations that are not present on the //c, then it will obviously not run properly and will have to be rewritten.)

It's not a two-way street, however. Assembly-language programs written specifically for the 65C02 may have to be partially translated before they will run on a computer using the 6502. This is because the 65C02 used on the //c supports ten new instructions and some new memory addressing techniques that the 6502 does not. If the 6502 is asked to interpret these new instructions, it will fail miserably.

In this chapter we will be taking special note of these new instructions. If you are writing software that must run on either a 65C02- or a 6502-based Apple, then you must not use them.

By the way, the “C” in 65C02 stands for CMOS, an acronym for Complementary Metal Oxide Semiconductor. This is the name for the process used to manufacture the transistors that form the 65C02 integrated circuit. A CMOS integrated circuit consumes far less power than a functionally identical circuit built using conventional technology. It will run cooler and can be operated by a smaller power supply. These are important factors when you want to design a small, portable, computer like the //c.

The 65C02 is an example of what is usually called an “8-bit” microprocessor. These types of microprocessors can handle data only one byte (8 bits) at a

time and they typically use 16 lines to address memory. Since each of these lines can be on or off, the 65C02 is capable of addressing 65,536 (2^{16}) memory locations at any given time. (Since one “K” of memory is equal to 1,024 bytes, this represents a “64K” memory space.) Contrast this with the newer wave of 16-bit microprocessors that can manipulate two bytes of data at once and have typical address spaces of one megabyte or more.

While the 65C02 is operating, it is continuously interpreting a stream of bytes in order to determine what it should do next. The bytes in this stream are controlled by the computer program that is being executed. This program contains instructions that enable the 65C02 to perform data transfers, input/output operations, logical operations, simple arithmetic, and other fundamental control operations.

In this chapter, we will take a brief look at the 65C02 instruction set and internal registers and describe how the 65C02 has been implemented on the //c. Note, however, that the purpose of this chapter is not to teach you 65C02 assembly-language programming, but rather to review some of the more important principles relating to the 65C02 microprocessor. Consult the references at the end of the chapter for a list of books that are available to teach you the art of programming the 65C02.

Important 65C02 Concepts

The 65C02 forms only one part of a microcomputer system such as the //c. The other important parts are the system memory (RAM and ROM) and the system input/output (I/O) devices. It is the 65C02, however, that is in charge of controlling both the accessing of memory and the passing of data to and from the I/O devices.

The 65C02 is told how and when to perform its chores by a series of instructions that it is constantly interpreting. These instructions will be discussed in the next section. In brief, they cause the 65C02 to perform a variety of data-manipulation tasks using a set of six internal registers that will be discussed below in the section entitled “65C02 Registers.”

Zero Page and the Stack

This is a convenient time to introduce you to two rather important areas of memory that are used in special ways by the 65C02 microprocessor: zero page and the stack.

Each 256 bytes of memory that starts at an address that is an integer multiple of \$100 (256), that is, \$0000, \$0100, \$0200, \$300, . . . , \$FF00 is called a “page” of memory. For example, the area of memory from \$BF00 through \$BFFF is referred to as page \$BF. Zero page, the page of memory from \$0000 . . . \$00FF, is treated in a special way by the 65C02. Generally speaking,

whenever the address on which a 65C02 instruction acts is contained in zero page, the highest two hexadecimal digits of the address do not have to be specified (since they are always zero by definition). This not only reduces the size of the program, it also allows the program to be executed more quickly. No wonder, then, that zero page is prime real estate as far as the 65C02 is concerned.

Page one of memory (\$100 . . . \$1FF) holds the 65C02 stack. The stack is used as a temporary data area by the 65C02 and several instructions can be used to implicitly read data from it or store data to it. These instructions are executed very quickly because they automatically calculate where to store the data or where to read it from by examining a special internal 65C02 “stack pointer” register. This register always points to the next free position available in the stack. When a byte is stored on the stack, it is stored at the position within page one given by the stack pointer and then the stack pointer is decremented by one. When a byte is removed from the stack, the stack pointer is incremented by one and then the byte is taken from the position within page one pointed to by the stack pointer.

We will be discussing the stack pointer, and other registers, in greater detail below.

65C02 Instruction Set

There are 66 general types of instructions that the 65C02 is capable of executing; they are listed in Table 2-1. These instructions include all 56 instructions supported by the standard 6502 microprocessor as well as 10 new ones used by the 65C02 only. The new instructions are marked with an asterisk in Table 2-1.

(The //c uses the version of the 65C02 produced by NCR Corporation. Another version, produced by Rockwell International Corporation, supports all of the NCR instructions and four additional ones called SMB (set memory bit), RMB (reset memory bit), BBS (branch on bit set), and BBR (branch on bit reset). You cannot use these instructions on the //c.)

Each instruction is actually a binary number that is interpreted by the 65C02 but it is usually represented by a three-character mnemonic name that is easier to remember. These mnemonics are used whenever an assembly-language program is being developed. The assembler that is used takes care of translating them into the corresponding binary numbers (the “machine language”) that the 65C02 can execute directly.

The 65C02 instructions can be used to perform a wide variety of functions. For example, they can be used to pass data between two registers or between registers and memory, to perform simple arithmetic, to increment and decrement index registers and memory locations, to pass data between registers and the stack, to perform logical functions, and so on. Figure 2-1 illustrates,

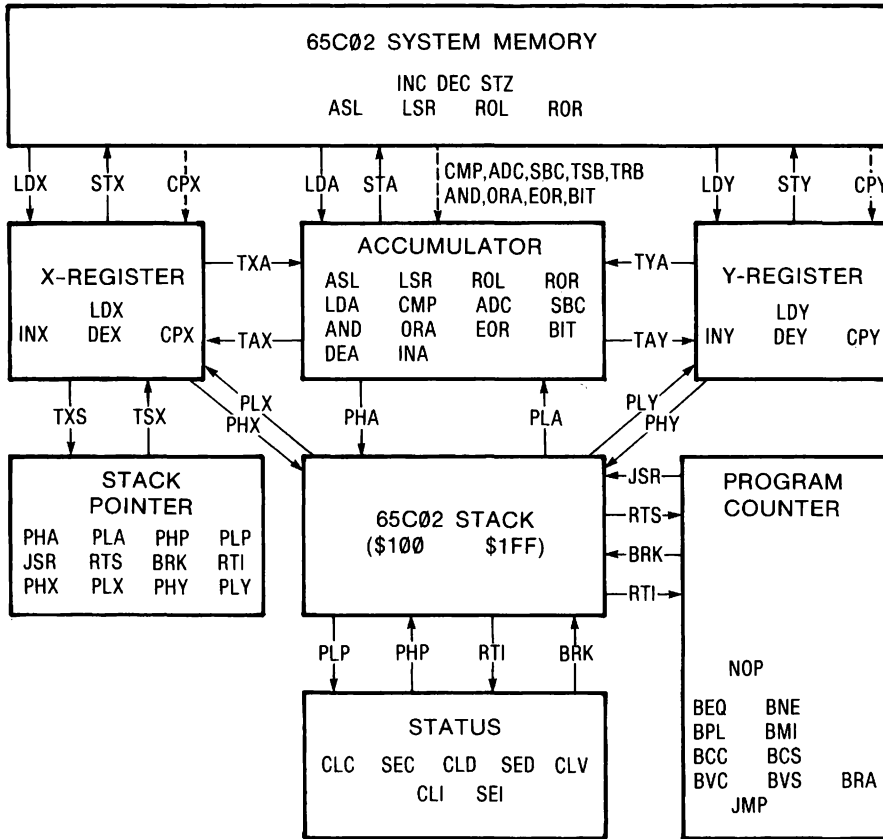
in a general way, how each of the 65C02's instructions affect memory and the 65C02 registers.

As you might expect, it takes a finite period of time for any particular instruction to be executed by the 65C02. The time required to execute one instruction, however, is not necessarily the same as the time required to

Table 2-1. 65C02 instruction set mnemonics in alphabetical order.

ADC	Add to accumulator	LDX	Load X register
AND	"And" with accumulator	LDY	Load Y register
ASL	Arithmetic bit-shift left	LSR	Logical bit-shift right
BCC	Branch on carry clear	NOP	No operation
BCS	Branch on carry set	ORA	"Or" with accumulator
BEQ	Branch on result zero	PHA	Push accumulator on stack
BIT	Test bits	PHP	Push status on stack
BMI	Branch on result minus	*PHX	Push X register on stack
BNE	Branch on result not zero	*PHY	Push Y register on stack
BPL	Branch on result plus	PLA	Pull accumulator from stack
*BRA	Branch relative always	PLP	Pull status from stack
BRK	Software interrupt	*PLX	Pull X register from stack
BVC	Branch on overflow clear	*PLY	Pull Y register from stack
BVS	Branch on overflow set	ROL	Rotate left through carry
CLC	Clear carry flag	ROR	Rotate right through carry
CLD	Clear decimal mode flag	RTI	Return from interrupt
CLI	Clear interrupt disable flag	RTS	Return from subroutine
CLV	Clear overflow flag	SBC	Subtract from accumulator
CMP	Compare with accumulator	SEC	Set carry flag
CPX	Compare with X register	SED	Set decimal mode flag
CPY	Compare with Y register	SEI	Set interrupt disable flag
*DEA	Decrement accumulator	STA	Store accumulator
DEC	Decrement memory by one	STX	Store X register
DEX	Decrement X register by one	STY	Store Y register
DEY	Decrement Y register by one	*STZ	Store zero in memory
EOR	"Exclusive-or" with accumulator	TAX	Transfer accumulator to X
*INA	Increment accumulator	TAY	Transfer accumulator to Y
INC	Increment memory by one	*TRB	Test and Reset with A
INX	Increment X register by one	*TSB	Test and Set with A
INY	Increment Y register by one	TSX	Transfer stack pointer to X
JMP	Jump to new location	TXA	Transfer X to accumulator
JSR	Jump + save return address	TXS	Transfer X to stack pointer
LDA	Load accumulator	TYA	Transfer Y to accumulator

*These instructions are not available on the 6502.



NOTE: Solid arrows indicate a transfer of data.
Dashed arrows indicate a transfer of information.

Figure 2-1. Usage chart of 65C02 instructions.

execute another. In fact, the time it takes to execute one general type of instruction will even vary depending on how the instruction is told to access the data on which it is to operate (that is, its “addressing mode”).

Table 2-2 sets out the times required to execute each instruction in units of 65C02 “machine cycles” for each valid addressing mode (addressing modes will be discussed in detail later in this chapter). The length of a 65C02 machine cycle is fixed by the frequency of the clock signal fed into the 65C02 microprocessor. On the //c, this clock signal is 1.023 megahertz, which means that every machine cycle takes 0.9775 (1/1.023) microsecond to perform.

It is often convenient to know exactly how long it will take to execute a particular instruction when precise timing loops must be generated in software. We will see an example of this in Chapter 9, where a program is presented that can generate musical notes of specific frequencies.

Table 2-2. 65C02 instruction set and cycle times.

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
ADC	#num	69	2	2	
	zpage	65	2	3	
	zpage,X	75	2	4	
	*(zpage)	72	2	5	
	(zpage,X)	61	2	6	
	(zpage),Y	71	2	5	(1)
	abs	6D	3	4	
	abs,X	7D	3	4	(1)
	abs,Y	79	3	4	(1)
AND	#num	29	2	2	
	zpage	25	2	3	
	zpage,X	35	2	4	
	*(zpage)	32	2	5	
	(zpage,X)	21	2	6	
	(zpage),Y	31	2	5	(1)
	abs	2D	3	4	
	abs,X	3D	3	4	(1)
	abs,Y	39	3	4	(1)
ASL	[accumulator]	0A	1	2	
	zpage	06	2	5	
	zpage,X	16	2	6	
	abs	0E	3	6	
	abs,X	1E	3	6	(3)
BCC	disp	90	2	2	(2)
BCS	disp	B0	2	2	(2)
BEQ	disp	F0	2	2	(2)
BIT	*#num	89	2	2	
	zpage	24	2	3	
	*zpage,X	34	2	4	
	abs	2C	3	4	
	*abs,X	3C	3	4	
BMI	disp	30	2	2	(2)
BNE	disp	D0	2	2	(2)
BPL	disp	10	2	2	(2)
BRA	*disp	80	2	2	(2)
BRK	[implied]	00	1	7	
BVC	disp	50	2	2	(2)
BVS	disp	70	2	2	(2)

(continued)

Table 2-2. 65C02 instruction set and cycle times (continued).

Instruction Mnemonic	Assembler Operand Format	Opcode Byte	Number of Bytes	Number of Clock Cycles	Notes
CLC	[implied]	18	1	2	
CLD	[implied]	D8	1	2	
CLI	[implied]	58	1	2	
CLV	[implied]	B8	1	2	
CMP	#num	C9	2	2	
	zpage	C5	2	3	
	zpage,X	D5	2	4	
	*(zpage)	D2	2	5	
	(zpage,X)	C1	2	6	
	(zpage),Y	D1	2	5	(1)
	abs	CD	3	4	
	abs,X	DD	3	4	(1)
	abs,Y	D9	3	4	(1)
CPX	#num	E0	2	2	
	zpage	E4	2	3	
	abs	EC	3	4	
CPY	#num	C0	2	2	
	zpage	C4	2	3	
	abs	CC	3	4	
DEA	*[accumulator]	3A	1	2	
DEC	zpage	C6	2	5	
	zpage,X	D6	2	6	
	abs	CE	3	6	
	abs,X	DE	3	6	(3)
DEX	[implied]	CA	1	2	
DEY	[implied]	88	1	2	
EOR	#num	49	2	2	
	zpage	45	2	3	
	zpage,X	55	2	4	
	*(zpage)	52	2	5	
	(zpage,X)	41	2	6	
	(zpage),Y	51	2	5	(1)
	abs	4D	3	4	
	abs,X	5D	3	4	(1)
	abs,Y	59	3	4	(1)
INA	*[accumulator]	1A	1	2	

(continued)

Table 2-2. 65C02 instruction set and cycle times (continued).

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
INC	zpage	E6	2	5	(3)
	zpage,X	F6	2	6	
	abs	EE	3	6	
	abs,X	FE	3	6	
INX	[implied]	E8	1	2	
INY	[implied]	C8	1	2	
JMP	abs	4C	3	3	(4)
	(abs)	6C	3	6	
	*(abs,X)	7C	3	6	
JSR	abs	20	3	6	
LDA	#num	A9	2	2	(1)
	zpage	A5	2	3	
	zpage,X	B5	2	4	
	*(zpage)	B2	2	5	
	(zpage,X)	A1	2	6	
	(zpage),Y	B1	2	5	
	abs	AD	3	4	
	abs,X	BD	3	4	
	abs,Y	B9	3	4	
LDX	#num	A2	2	2	(1)
	zpage	A6	2	3	
	zpage,Y	B6	2	4	
	abs	AE	3	4	
	abs,Y	BE	3	4	
LDY	#num	A0	2	2	(1)
	zpage	A4	2	3	
	zpage,X	B4	2	4	
	abs	AC	3	4	
	abs,X	BC	3	4	
LSR	[accumulator]	4A	1	2	(3)
	zpage	46	2	5	
	zpage,X	56	2	6	
	abs	4E	3	6	
	abs,X	5E	3	6	
NOP	[implied]	EA	1	2	
ORA	#num	09	2	2	
	zpage	05	2	3	
	zpage,X	15	2	4	
	*(zpage)	12	2	5	

(continued)

Table 2-2. 65C02 instruction set and cycle times (continued).

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
	(zpage,X)	01	2	6	
	(zpage),Y	11	2	5	(1)
	abs	0D	3	4	
	abs,X	1D	3	4	(1)
	abs,Y	19	3	4	(1)
PHA	[implied]	48	1	3	
PHP	[implied]	08	1	3	
PHX	*[implied]	DA	1	3	
PHY	*[implied]	5A	1	3	
PLA	[implied]	68	1	4	
PLP	[implied]	28	1	4	
PLX	*[implied]	FA	1	4	
PLY	*[implied]	7A	1	4	
ROL	[accumulator]	2A	1	2	
	zpage	26	2	5	
	zpage,X	36	2	6	
	abs	2E	3	6	
	abs,X	3E	3	6	(3)
ROR	[accumulator]	6A	1	2	
	zpage	66	2	5	
	zpage,X	76	2	6	
	abs	6E	3	6	
	abs,X	7E	3	6	(3)
RTI	[implied]	40	1	6	
RTS	[implied]	60	1	6	
SBC	#num	E9	2	2	
	zpage	E5	2	3	
	zpage,X	F5	2	4	
	*(zpage)	F2	2	5	
	(zpage,X)	E1	2	6	
	(zpage),Y	F1	2	5	(1)
	abs	ED	3	4	
	abs,X	FD	3	4	(1)
	abs,Y	F9	3	4	(1)
SEC	[implied]	38	1	2	
SED	[implied]	F8	1	2	

(continued)

Table 2-2. 65C02 instruction set and cycle times (continued).

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
SEI	[implied]	78	1	2	
STA	zpage	85	2	3	
	zpage,X	95	2	4	
	*(zpage)	92	2	5	
	(zpage,X)	81	2	6	
	(zpage),Y	91	2	5	(1)
	abs	8D	3	4	
	abs,X	9D	3	4	(1)
STX	abs,Y	99	3	4	(1)
	zpage	86	2	3	
	zpage,Y	96	2	4	
STY	abs	8E	3	4	
	zpage	84	2	3	
	zpage,X	94	2	4	
STZ	abs	8C	3	4	
	*zpage	64	2	3	
	*zpage,X	74	2	4	
	*abs	9C	3	4	
TAX	*abs,X	9E	3	5	
	[implied]	AA	1	2	
TAY	[implied]	A8	1	2	
TRB	*zpage	14	2	5	
	*abs	1C	3	6	
TSB	*zpage	04	2	5	
	*abs	0C	3	6	
TSX	[implied]	BA	1	2	
TXA	[implied]	8A	1	2	
TXS	[implied]	9A	1	2	
TYA	[implied]	98	1	2	

Instructions marked with an asterisk are not available on the 6502.

Notes:

- (1) Add one clock cycle if a page boundary is crossed.
- (2) Add one clock cycle if a branch occurs to a location in the same page; add two clock cycles if a branch occurs to a location in a different page.
- (3) Add one clock cycle if a page boundary is crossed; always 7 cycles on the 6502.
- (4) 5 cycles on the 6502.

See Table 2-3 for a description of the assembler operand formats.

65C02 Registers

While the 65C02 is executing a program, it makes use of the six internal registers that are shown in Figure 2-2. These registers are used to manipulate data in the manner dictated by the program that is executing and also to make the 65C02 aware of various aspects of the status of the system: where the next instruction to be executed is located, where the next free space in the stack is located, and what the status of its seven internal flags is. A detailed understanding of these registers is important before a 65C02 assembly-language program can be written. We will now take a closer look at each of the six registers.

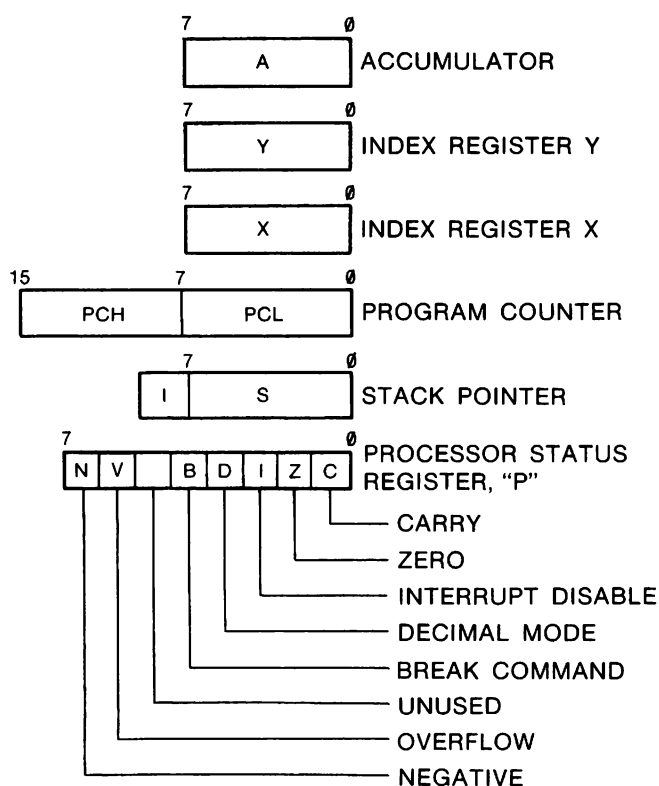


Figure 2-2. The 65C02 registers.

The Accumulator—A

The 65C02 supports two simple arithmetic instructions: ADC (add with carry) and SBC (subtract with carry). Both of them require that the first of

the two operands in the addition or subtraction be contained in the accumulator register, A. After the arithmetic has been performed, the result is stored in A. This is how it gets its name—it “accumulates” the results of arithmetic operations that are performed. The accumulator is an 8-bit register and so can hold numbers from 0 to 255 only.

The accumulator is unique in that it is the only one of the 65C02's registers that can be used to perform the logical instructions, namely, EOR (logical “exclusive-or”), ORA (logical “or”), and AND (logical “and”), or any of the bit-shifting instructions, namely, ASL (arithmetic shift left), LSR (logical shift right), ROL (rotate left), and ROR (rotate right). (You should note, however, that the bit-shifting instructions can also operate directly on memory locations.)

Here are the 65C02 instructions that directly use and affect the accumulator:

- Arithmetic : ADC, SBC
- Increment : INA (not on 6502)
- Decrement : DEA (not on 6502)
- Logical : AND, ORA, EOR, BIT, TRB (not on 6502), TSB (not on 6502)
- Bit-shifting : ASL, LSR, ROL, ROR
- Compare : CMP
- Store in memory : STA
- Load from memory or with data : LDA
- Store on stack : PHA
- Load from stack : PLA
- Inter-register transfer : TAX, TAY, TXA, TYA

The Index Registers—X and Y

Like the accumulator, the X and Y index registers are eight bits in size and can contain numbers from 0 to 255.

As their name suggests, the index registers are used primarily to locate elements contained in data structures in memory, such as a series of elements in a one-dimensional array. This is done by fixing the beginning address of the data structure and then simply adjusting the index register so that the sum of the beginning address and the index register is equal to the address of the element that is to be accessed.

The 65C02 supports several special instructions that directly use and affect the index registers:

- Increment : INX, INY
- Decrement : DEX, DEY

- Inter-register transfer : TAX, TAY, TXA, TYA, TXS, TSX
- Store in memory : STX, STY
- Store on stack : PHX (not on 6502), PHY (not on 6502)
- Load from stack : PLX (not on 6502), PLY (not on 6502)
- Load from memory or with data : LDX, LDY
- Compare : CPX, CPY

Note that the logical instructions and bit-shifting instructions that can be used with the accumulator cannot be used with the index registers.

The Processor Status Register—P

The 8-bit processor status register holds the states of seven one-bit flags or “status” bits that are referenced by the 65C02 when it is executing many of its instructions. (One bit in the processor status register, bit 5, is not used by the 65C02.) Each of these flags has a specific meaning and can markedly affect how instructions are executed. For example, the 65C02 supports a series of “branch on condition” instructions (BCC, BCS, BPL, BMI, BEQ, BNE, BVC, BVS), each of which can be used to examine the status of a particular flag and to cause the program to “jump” to a new location if the condition is met or to continue on with the next instruction in memory if it is not. (There is also a “branch always” instruction, BRA, that will cause an unconditional jump.)

Although almost all instructions will cause flags in the processor status register to be adjusted after they have been executed, the following instructions explicitly affect them:

- Clear and set the carry flag : CLC, SEC
- Clear and set the decimal flag : CLD, SED
- Clear and set the interrupt flag : CLI, SEI
- Clear the overflow flag : CLV

Let’s take a look at each of these seven flags right now.

Carry Flag (C)

The 65C02 uses the carry flag in three quite different ways.

First, the carry flag represents the “ninth” bit in any unsigned addition (ADC) or subtraction (SBC) operation that is performed. (“Unsigned” means that all eight bits of a byte are used to represent the magnitude of a number.) It can be examined after the addition or subtraction in order to determine whether the result is outside the range of numbers that can be stored in the 8-bit accumulator. This allows for easy manipulation of numbers that use more than one byte.

The 65C02 can perform arithmetic in one of two modes: binary and decimal. The mode used depends on the setting of the status register's decimal mode flag (see below).

In binary mode, each byte is considered to represent a simple unsigned binary number from 0 . . . 255. When arithmetic operations are performed, the standard rules for adding or subtracting two binary numbers are followed.

In decimal mode, however, each half of the byte is considered to represent a single decimal digit from 0 to 9; this means that only those decimal numbers from 00 . . . 99 can be represented. When arithmetic operations are performed on such numbers, the result is always stored in the same decimal format.

In either mode, before any arithmetic is performed, the carry flag must be cleared with a CLC instruction, in the case of addition, or set with a SEC instruction, in the case of subtraction. (If multibyte arithmetic is being performed, then the carry is adjusted only at the beginning of the sequence of additions or subtractions.) If the state of the carry flag changes after an addition operation, then the true answer is 256 (if in binary mode) or 100 (if in decimal mode) more than the number in the accumulator. If the carry flag changes after a subtraction, then the true answer is 256 (if in binary mode) or 100 (if in decimal mode) less than the number in the accumulator.

The second use of the carry flag is as a ninth bit that participates whenever the ASL, LSR, ROL, and ROR bit-shifting instructions are executed.

Third, the carry flag is used as a general-purpose flag that is acted on by the BCC (branch if C-flag is clear, or 0) and BCS (branch if C-flag is set, or 1) instructions. As with all of the 65C02's "branch on condition" instructions, BCC and BCS allow control of the program flow to be manipulated by the state of a flag in the processor status register (in this case, the carry flag).

Zero Flag (Z)

This flag is used to indicate whether the last data movement or arithmetic operation involved a zero result. If it did, then the Z-flag will be set (1); otherwise it will be cleared (0).

There are two branch instructions that examine the status of the Z-flag to determine whether the branch should be performed: BEQ (branch if Z-flag is 1, that is, result was equal to zero) and BNE (branch if Z-flag is 0, that is, result was not equal to zero).

Interrupt Disable Flag (I)

This flag is used to control how the 65C02 will react when the electrical signal on its IRQ (interrupt request) pin is brought near 0 volts. Such an interrupt can be generated by the Apple mouse or the //c's two serial ports whenever they are ready to send information to, or receive information from,

the //c. If the I-flag is set using the SEI instruction, then all IRQ signals that may be generated will be ignored. If, however, the I-flag is cleared using the CLI instruction, then the 65C02 will respond to IRQ signals when they occur by beginning a special interrupt sequence that is described in detail below in the section entitled "65C02 Interrupts."

Decimal Mode Flag (D)

This flag is used to control how the 65C02 is to perform addition and subtraction operations. If standard binary arithmetic is to be performed using the ADC and SBC instructions, then this flag must be cleared to 0 using the CLD instruction. As we saw when discussing the accumulator, in binary mode bytes are treated as unsigned binary numbers from 0 to 255.

If, however, the D-flag is set to 1 using the SED instruction, all arithmetic will be performed under the assumption that all numbers are stored in a special decimal format. In this format, one byte is used to store exactly two decimal digits from 0 to 9. The first digit is stored in the high-order four bits and the other in the low-order four bits and the maximum number that can be stored is 99. When arithmetic operations are performed, the results will also be stored in this format.

Break Flag (B)

This flag is adjusted internally by the 65C02 whenever an IRQ (interrupt request) interrupt is recognized by the 65C02 or a BRK (break) instruction is executed. See the section below entitled "65C02 Interrupts" for more information on these types of interrupts. When an IRQ interrupt is recognized, then the B-flag is cleared to 0; if a BRK instruction is executed, then it is set to 1.

Whenever an IRQ or a BRK interrupt is generated, the 65C02 begins to execute the same program (its address is held at locations \$FFFE and \$FFFF). It is often convenient, however, to determine what the source of the interrupt was so that a different action can be taken for each source. This is most easily done by having the interrupt-servicing program examine the state of the B-flag.

Overflow Flag (V)

The overflow flag is used primarily when performing arithmetic operations on signed numbers. Signed numbers are those that use bit 7 of a byte to hold the sign of the number (1 for negative, 0 for positive). Bits 0 . . . 6 are used to store the magnitude of the number in a special "two's complement" format that will be described in Chapter 4. If the result of an addition or subtraction of two signed numbers is outside the range of numbers that can be stored in

this format ($-128 \dots +127$), then the V-flag will be set to 1; if the number is in range, however, the V-flag will be cleared to 0.

The V-flag can be explicitly cleared by using the CLV instruction. Surprisingly, there is no corresponding instruction to explicitly set the V-flag.

The state of the V-flag can also be affected by using the BIT instruction. If you “BIT” any memory location, then a copy of bit 6 of the byte stored there will be placed in the V-flag.

Two branch instructions make use of the V-flag: BVS (branch if V-flag is 1) and BVC (branch if V-flag is 0).

Negative Flag (N)

The negative flag is used to indicate the sign of the last value that was directly transferred into the A, X, or Y register or that was put there by an instruction that performed an arithmetic operation (DEX, DEY, INX, INY, ADC, SBC, and so on). The 65C02 considers any byte that contains a one in bit 7 to be negative.

Two branch instructions make use of the N-flag: BPL (branch on plus, that is, N-flag is 0) and BMI (branch on negative, that is, N-flag is 1).

A BIT instruction can also be used to directly affect the state of the N-flag. When you “BIT” any memory address, a copy of bit 7 of the byte stored there will be placed in the N-flag. If bit 7 is used to hold the status of some condition, then you can use BPL to branch if the status is off (0) or BMI to branch if it is on (1). We will see in later chapters that the //c uses bit 7 of several locations to represent the status of different hardware switches that can be controlled by software.

The Stack Pointer—S

As we saw earlier in this chapter, the 65C02 uses the 256-byte area from \$100 to \$1FF as a hardware stack. This is a “last-in, first-out” data area: the most recent information stored on the stack is always removed first. Information is usually placed on the stack by the “push” instructions, PHA, PHP, PHX, and PHY, and removed from the stack by the “pull” instructions, PLA, PLP, PLX, and PLY. (Information does not actually disappear after a pull, but it will be overwritten as soon as more information is pushed on to the stack.)

The JSR (jump-to-subroutine) instruction also causes information to be placed on the stack. When the JSR instruction is executed, the address of the next instruction in memory after the JSR, minus one, is pushed on the stack (high-order byte first). When the corresponding RTS (return-from-subroutine) instruction is executed, this address is removed and the program resumes at that address (plus 1).

The stack pointer register, S, is used to keep track of where in the 256-byte stack area the bytes are to be pushed to or pulled from; it always points to the next free space available in the stack area. When the system is first initialized, S is set equal to \$FF. Then, whenever a byte is pushed on the stack, it is stored at location $\$100 + S$ and then the stack pointer is decremented by one. Because S is decremented, the stack grows downward in memory. When bytes are pulled from the stack, they are taken from the top of the stack (location $\$100 + S + 1$). The stack pointer is automatically incremented each time a byte is removed from the stack in this way.

Interrupt conditions and interrupt-related instructions also affect the stack pointer (see the section below entitled “65C02 Interrupts” for a detailed discussion of interrupts). When an interrupt is recognized, a two-byte address and a copy of the processor status register is placed on the stack and the stack pointer is decremented by three. When the corresponding RTI (return-from-interrupt) instruction is executed, the three bytes on top of the stack will be placed in the status register and the program counter and the stack pointer will be incremented by three.

Here are the 65C02 instructions that directly affect the stack pointer register:

- Inter-register transfer : TXS, TSX
- Push data on stack : JSR, PHA, PHP, PHX, PHY, BRK
- Pull data from stack : PLA, PLP, PLX, PLY, RTS, RTI

The Program Counter—PC

The program counter (sometimes called the instruction pointer) is the only 16-bit register that the 65C02 supports and is used to hold the address of the next instruction to be executed. This address will normally be that of the next instruction in the program, but not necessarily. There are several instructions that can be used to manipulate the flow of the program and to pass control to other parts of the program by adjusting the program counter accordingly. These are the JMP (jump) instruction, which acts like an Applesoft GOTO, the JSR (jump-to-subroutine) and RTS (return-from-subroutine) instructions, which act like an Applesoft GOSUB/RETURN combination, the branch-on-condition instructions (BCC, BCS, BEQ, BNE, BPL, BMI, BVC, BVS), and the branch always (BRA) instruction. The program counter is also affected by any hardware or software interrupt (BRK) and by the RTI (return-from-interrupt) instruction.

65C02 Addressing Modes

A complete 65C02 instruction is either one, two, or three bytes long. The first byte always represents the operation code (“opcode”) for the instruction

itself and the remaining bytes (if any) represent the operand; if an operand is specified, it is either an address (one byte or two bytes) or immediate data (one byte). If the operand represents a two-byte address, then the first byte is always the lower two digits of the four-digit hexadecimal address (the allowable addresses are in the range \$0000 to \$FFFF).

An address that is specified after an opcode is not necessarily the address from which the instruction will read data or to which it will store data. In many instances, the 65C02 uses this address to calculate another address (called the “effective address”) on which it does operate. Exactly how this calculation is to be performed depends on which of several “addressing modes” that can be used by that instruction has been selected. The 65C02 determines which addressing mode has been selected by examining the value of the opcode itself—each general type of instruction can have several opcode values associated with it, one for each valid addressing mode. The value of the opcode also dictates whether the operand is to be interpreted as immediate data instead of an address.

We will now outline the various addressing modes that the 65C02 supports. Before beginning, you should note that not all instructions are permitted to use each addressing mode. The ones that are supported by each instruction are indicated by entries in Table 2-2. The names of each of the addressing modes that the 65C02 uses, and the operand formats used to represent these modes in an assembly-language program, are summarized in Table 2-3. Note that these operand formats are those used by the Merlin Pro assembler that was used to develop the examples presented in this book; other assemblers may require that slightly different formats be used.

Immediate

Immediate addressing is used whenever you want an instruction to act on a specific 8-bit number provided by the program rather than on a byte stored somewhere in memory. This 8-bit number is stored in the byte immediately following the opcode itself and forms the operand for the instruction.

The immediate addressing mode is most useful for initializing a register to a constant value and for providing specific data on which an instruction is to operate. To select this addressing mode when using an assembler, the “#” symbol must be placed in front of the number in the instruction’s operand:

```
LDA #49    load the accumulator with 49 (decimal)
LDX #$43   load X with $43 (hexadecimal)
```

It is often necessary to deal with the high-order or low-order byte of a two-byte address as an immediate quantity. To do this, you must use an assembler operand of the form “#<ADDRESS” (for the low-order byte) and “#>ADDRESS” (for the high-order byte), where “ADDRESS” is the address being dealt with. Note, however, that the form of this type of operand applies to the Merlin Pro assembler only; most other assemblers require that a dif-

Table 2-3. 65C02 addressing modes and assembler operand formats.

Addressing Mode	Assembler Operand Format	Example of Instruction
Immediate	#num	LDA #\$45
	#<abs	LDA #<\$FD1B
	#>abs	LDA #>\$FD1B
Absolute	abs	LDX \$FE44
	zpage	LDA \$24
Accumulator	[Not applicable]	ASL
Implied	[Not applicable]	CLC
Indexed indirect	(zpage,X)	LDA (\$E0,X)
	(abs,X)	JMP (\$2000,X)
Indirect indexed	(zpage),Y	STA (\$28),Y
Absolute indexed	abs,X	LDA \$2000,X
	abs,Y	STA \$0400,Y
	zpage,X	LDA \$28,X
	zpage,Y	STX \$22,Y
Relative*	disp	BNE \$BEAF
Absolute Indirect	(abs)	JMP (\$03EE)
	(zpage)	STA (\$E0)

Notes: "num" = 1-byte number
 "abs" = 2-byte address
 "<abs" = low-order byte of a 2-byte address (or constant)
 ">abs" = high-order byte of a 2-byte address (or constant)
 "zpage" = 1-byte zero page address
 "disp" = 1-byte signed displacement

*Relative addressing: An absolute address is usually specified in the operand in the program source code; the assembler converts the operand to a one-byte displacement to this address when the program is assembled.

ferent method be used to specify which half of an address is to be dealt with. One assembler, the Apple 6502 Editor/Assembler, uses the same general method, but it reverses the meaning: "#>" is used to specify the low-order byte and "#<" is used to specify the high-order byte!

Absolute

The absolute addressing mode is used whenever the operand itself contains the address in memory on which the opcode is to operate. The two bytes required to store this address are stored low-byte first.

Here are some examples of how to use the absolute addressing mode:

```
LDA $FE43    load the accumulator with the number stored at $FE43
STY $1238    store the Y register at location $1238
```

Some instructions support an important variant of the absolute addressing mode, called zero page absolute, if the address specified is in the 65C02 zero page (the first 256 bytes of memory). This mode is identified by a different opcode byte. In this mode, the opcode is followed by a one-byte address only because the high-order byte is implicitly zero. Most assemblers will recognize when a zero page location is being specified and will automatically select this addressing mode for you by changing the value of the opcode byte used by the instruction when the program is assembled.

Accumulator

Accumulator addressing is the mode used by all those opcodes that act on the accumulator alone and that require no address or immediate data on which to operate. These are the DEA and INA instructions and the bit-shifting instructions LSR, ASL, ROL, and ROR. There are no operand bytes for these instructions. Note, however, that some assemblers other than Merlin Pro (notably, the Apple 6502 Editor/Assembler) require that the letter “A” be entered in the operand field before the program source code can be properly assembled.

Implied

The 65C02 supports many opcodes that do not act on immediate data or on memory locations, but rather on internal registers and status flags only. These opcodes require no operands because their actions are implicitly defined by the opcode itself and so the addressing mode used is called implied.

Here are some examples of opcodes that use the implied addressing mode: PHA, PLA, PHP, PLP, CLD, CLI, BRK, DEX, INX, NOP, RTS, TAX.

Zero-Page Indexed Indirect

When the zero-page indexed indirect addressing mode is used, the operand is only one byte long and represents a location in zero page. The effective address on which the instruction acts is calculated by first adding the contents of the X register to the zero page location specified in the operand to obtain a resultant address. The effective address is represented by the two bytes that are stored at the resultant zero-page address and the very next address (low-order byte first).

For example, if X is 3 and the zero-page address is \$E0, then the effective address is stored at \$E0 + 3 (low-order part) and \$E0 + 4 (high-order part).

You can select this addressing mode when using an assembler by using an instruction of the form

```
STA ($E0, X)
```

where the parentheses indicate that the effective address is not \$E0 + X but rather the address stored at that location.

Zero-Page Indirect

The zero-page indirect addressing mode is unique to the 65C02 and is not available on the 6502. The operand is one byte long and represents a location in zero page. The effective address is simply the address that is stored at this location and the very next one. Thus the zero-page indirect mode is the same as the zero-page indexed indirect mode, but without the X indexing.

An assembler uses an instruction of the form

```
STA ($E0)
```

to indicate that the zero-page indirect mode is to be used.

Indirect Indexed

Indirect indexed is a powerful addressing mode that is often used to access a block of memory that may not always begin at the same location in memory or that is longer than 256 bytes in length. The operand is one byte long and represents a zero page location; this zero page location, and the one immediately following it, contain the address (low-byte first) of the beginning of a data block in memory. These locations are said to “point to” this data block.

When this addressing mode is used, the effective address on which the instruction is to operate is calculated by first taking the address of this data block from the zero page locations and then adding to it the contents of the Y register.

Here is an example of how you would select the indirect indexed addressing mode when using an assembler:

```
LDA ($26), Y
```

The parentheses around \$26 mean “contents of”; it is the address stored at \$26 (and \$27) that will be used to calculate the effective address, and not \$26 itself. If the Y-register contains \$FE and the address \$400 is stored at \$26/\$27 (\$00 in location \$26 and \$04 in location \$27), then the accumulator will be loaded with the contents of memory location \$4FE (\$4FE = \$400 + \$FE).

Absolute Indexed

The operand for the absolute indexed addressing mode is two bytes long and contains the absolute address of a memory location called a “base address.” The effective address on which the instruction is to operate is calculated by taking this base address and adding to it the contents of the X register (if X indexing is selected) or of the Y register (if Y indexing is selected).

Here are some examples of the use of this addressing mode:

LDA \$400, X	load the accumulator with the contents of the location specified by \$400 + X.
STA \$A032, Y	store the accumulator at the location specified by \$A032 + Y

There is a special version of this addressing mode, called zero page absolute indexed, that can be used by some instructions when the base address is in page zero. In this case, the operand is only one byte long and represents this zero page address. Most assemblers will automatically select this addressing mode for you if the operand is, indeed, in page zero.

Relative

The 65C02 supports a series of two-byte branch instructions that examine the 65C02 status register to determine whether a change in the flow of the program should be made or not: BEQ, BNE, BPL, BMI, BCC, BCS, BVC, and BVS. Another instruction, not available on the 6502, BRA, can be used to unconditionally change the flow of the program.

The first byte represents, as usual, the opcode for the instruction. The second byte represents the number that must be added to the address of the next instruction in memory in order to calculate the destination address of the branch. Because this byte represents a displacement from an instruction’s location rather than an absolute location, this addressing mode is called “relative.”

There are restrictions on how far you can branch using relative addressing. In particular, you can only specify a relative address that is at most 127 bytes higher in memory or 128 bytes lower in memory (as measured from the address of the next higher instruction). Values from \$00 . . . \$7F represent the positive branches (0 . . . 127), and values from \$80 . . . \$FF represent the negative branches (−128, −127, . . . , −1). Note that the values for negative branches are stored in a special “two’s complement” format; see Chapter 4 for a detailed description of this format.

If you must transfer control to a destination location that is outside this range, you will have to use a JMP instruction instead.

Absolute Indirect

This addressing mode is used by only one instruction, **JMP**. A two-byte operand is used and these two bytes define a location in memory that contains the low half of the address that is to be jumped to; the high half is stored in the next memory location.

If you are using an assembler, then you would select this addressing mode by entering an instruction that looks like this:

```
JMP ( $1234 )
```

The parentheses around the operand indicate that it is not \$1234 that is being jumped to but rather the address stored at \$1234 (and \$1235).

The absolute indirect addressing mode is useful in situations where the ultimate destination of the jump instruction may be changed, perhaps by another program. Even if this other program places a new address at the operand address, the main program itself need not be changed. On the other hand, if the absolute addressing mode were used instead, then it would be necessary to modify the program and this may be difficult to do. The *//c* uses the indirect addressing mode whenever it has to jump to its character input or output subroutines. Whenever new input or output devices are activated, all that need be done is to change the address stored at the address specified in the operand—the main program will remain the same (see the discussion of the *//c*'s input and output links in Chapters 6 and 7).

Absolute Indexed Indirect

The absolute indexed indirect addressing mode is the other mode that is not available on the 6502. It works with the **JMP** instruction only and is of the form:

```
JMP ( $1234 , X )
```

The effective address that is jumped to is the one stored beginning at an address that is equal to the address specified in the operand plus the contents of the X-register.

65C02 Input/Output Handling

Unlike the instruction sets of some microprocessors, the 65C02 instruction set does not include any instructions that are specifically designed to perform input/output (I/O) operations like reading from the keyboard or writing to the video screen. Instead, all I/O operations are performed by using standard instructions to read data from or write data to addresses within the 65C02's standard 64K address space to which I/O devices are "connected." These addresses do not usually represent real RAM or ROM memory locations

(memory that holds video display information is one exception) but, nevertheless, are accessed in exactly the same way as if they did.

This method of handling I/O is called “memory-mapped I/O” because the I/O devices form a logical part of the 65C02’s 64K memory space itself and so no special instructions are required to make use of them. The //c contains several addresses that are used to control various aspects of its hardware environment. As we will see at the end of this chapter, except for those addresses that relate to the video display, these addresses are all mapped to locations from \$C000 . . . \$C0FF. Note that some of these I/O locations can be accessed in order to switch between one of two hardware states, for example, text or graphics display, primary or alternative character set, and 40-column or 80-column display. Thus, they are called “soft switch” I/O memory locations.

65C02 Interrupts

There are three input pins on the 65C02 integrated circuit that are called RESET, IRQ (interrupt request), and NMI (non-maskable interrupt). When the electrical signals at each of these three pins is high (near +5 volts) the 65C02 goes about performing its normal functions. If, however, one of these pins is suddenly brought low (near 0 volts), one of three special “interrupt” sequences may begin, depending on which pin has been affected. An interrupt sequence can also be generated in software by using the 65C02 BRK (break) instruction.

We won’t concern ourselves with NMI interrupts because they are not used on the //c.

RESET interrupts can be generated in three different ways on the //c: when the power is turned on, when [control-RESET] is pressed, and when [control-OPEN-APPLE-RESET] is pressed. (See Chapter 6.) RESET signals are usually used to initialize the system to its power-on state or to break out of a running program.

IRQ interrupts are usually generated by I/O devices whenever they have information available to be read (input devices) or whenever they are ready to receive information (output devices). Generally speaking, when the 65C02 detects an IRQ signal, it stops executing the main program and starts executing a special interrupt-handling subroutine that has been installed. When this subroutine finishes, control returns to the main program at the point where it was interrupted and execution of that program continues.

Since the 65C02 can be interrupted like this, it is not necessary for the main program to continuously monitor (or “poll”) the I/O devices to determine when one is ready to be dealt with. This means that the program can execute much more quickly and efficiently.

IRQ interrupts may be generated on the //c in any of the following circumstances:


- When the mouse is moved. (See Chapter 10.)
- When a video vertical blanking signal occurs. (This occurs every 1/60th of a second; see Chapter 10.)
- When a serial port is ready to send or receive data. (See Chapter 11.)
- When the state of the signal on pin 5 of either of the two serial ports changes. (This is the serial device's "data carrier detect" (DCD) line; see Chapter 11.)
- When a key is pressed or released. (See Chapter 11.)
- When the state of the signal on pin 9 of the external disk drive connector changes. (This is called the "external interrupt" line; see Chapter 11.)

Each type of 65C02 interrupt is associated with a two-byte vector that holds the address of the interrupt-handling subroutine that will be called when the interrupt occurs. These vectors are all stored in the high end of the 65C02 memory space from \$FFFA to \$FFFF. The specific vector locations for each type of interrupt and the addresses of the interrupt-handling routines to which they point are shown in Table 2-4. Note that all of the vector addresses change when ProDOS is being used. Most of ProDOS resides in a special "bank-switched RAM" area from \$D000 . . . \$FFFF that is normally occupied by Applesoft and the system monitor ROMs (see Chapter 8). Thus, the interrupt vectors within this RAM area (from \$FFFA to \$FFFF) can be changed as desired and they will take effect whenever bank-switched RAM is active.

The interrupt-handling routines on the //c ultimately pass control to other addresses that are specified in user-definable vector locations. These user vector locations are also shown in Table 2-4. Note that a user-defined interrupt

Table 2-4. 65C02-Apple //c interrupt locations.

<i>Interrupt Type</i>	<i>Interrupt Vector Location</i>	<i>Address of Interrupt Handler</i>	<i>Location of User Vector</i>
RESET	\$FFFC/\$FFFD	\$FA62 or \$FFCB	\$03F2*
IRQ	\$FFFE/\$FFFF	\$C803 or \$FF9B	\$03FE
BRK	\$FFFE/\$FFFF	\$C803 or \$FF9B	\$03F0



when the ProDOS
bank-switched
RAM area is active
only

*Control is passed to the Reset user vector only if the number stored at \$3F4 (the powered-up byte) is equal to the logical exclusive-OR of the number stored at \$3F3 and the constant \$A5.

subroutine that is used to handle interrupts generated by an IRQ signal, or a BRK command, must end by executing an RTI (return-from-interrupt). Failure to do this will cause the system to crash in an unpredictable way.

The three basic types of interrupts supported by the 65C02 on the Apple //c will now be discussed in detail.

Reset Interrupt

The reset interrupt is used to cause the system to stop executing the current program and to begin a sequence of instructions that start at the address stored in the reset vector at \$FFFC/\$FFFD (low-order byte first). On the //c, the reset vector points to a subroutine beginning at \$FA62 (the ProDOS reset vector actually points to another subroutine that ultimately calls \$FA62). This subroutine takes care of initializing the //c to a known state and will pass control to a user-definable subroutine whose address is stored at \$3F2 and \$3F3 (low-order byte first) if the logical exclusive-OR of the value stored at \$3F3 and the constant \$A5 is the same as the value stored at \$3F4 (which is called the powered-up byte). If it isn't, then the disk drive will start up just as it does when the //c is first turned on.

The reset interrupt is automatically generated whenever the power to the 65C02 is first turned on. As we will see in Chapter 6, it can also be generated by pressing the CONTROL and RESET keys on the //c's keyboard at the same time. Specific examples of "trapping" the reset interrupt by adjusting the user vector at \$3F2/\$3F3 and the powered-up byte at \$3F4 will also be given in Chapter 6.

A reset interrupt is normally used only in panic situations where the program that is running must be stopped immediately or when you are running programs that do not have an exit command.

Interrupt Request (IRQ)

The 65C02 will only respond to an active IRQ interrupt signal if the I-flag in the processor status register is 0 (this flag is cleared using the CLI instruction). If the I-flag is set to 1 (using the SEI instruction), then the IRQ signal will be ignored and no further IRQ interrupts will be dealt with until after the I-flag is cleared.

If the I-flag is 0 and an active IRQ signal is generated, then the 65C02 responds by first completing the current instruction being executed. The following sequence of events then takes place:

- The current program counter is stored on the stack. (This will be the address, less 1, of the next instruction in the program to be executed after the interrupt has been dealt with.)

- The B-flag in the processor status register is cleared to 0 and then the register is stored on the stack.
- The I-flag in the processor status register is set to 1. (This disables subsequent IRQ operations until the current interrupt is dealt with; see below.)

After these operations have been performed, the program counter is loaded with the address that is stored in the IRQ vector at \$FFFE/\$FFFF (low-order byte first), and then the interrupt-handling program that begins at that address is executed. The address stored in the IRQ vector on the //c is usually \$C803, or if ProDOS is processing data when the interrupt occurs, at \$FF9B; the ProDOS subroutine simply does a bit of housecleaning before calling \$C803 itself.

The subroutine beginning at \$C803 first determines whether the source of the interrupt was an IRQ signal or a BRK instruction (BRK is discussed in the next section). If the source was an IRQ signal, the subroutine calls two ROM subroutines that are used to handle mouse port and serial port interrupts. These subroutines will either handle the interrupt internally or pass control through to your own interrupt-handling subroutine, depending on the values stored in certain memory locations. We will see how to manipulate mouse port and serial port interrupts in Chapters 10 and 11.

If the interrupt is not handled internally by the //c, then control will pass to the address stored at user vector locations \$3FE and \$3FF. Thus, to properly handle an IRQ interrupt, an interrupt-handling subroutine must be placed in memory and its starting address must be stored at \$3FE/\$3FF. This subroutine must pass control back to the main program by ending with an RTI (return from interrupt) instruction. (Alternatively, if ProDOS is being used, you can install your interrupt-handling routine by using a special ProDOS interrupt command. See Apple's ProDOS Technical Reference Manual for more information on this feature of ProDOS.)

The BRK Instruction

One of the 65C02's instructions allows you to simulate the effect of an IRQ signal in software. This is the one-byte BRK (break) instruction represented by a "00" byte. BRK is primarily used when debugging a program because when a program encounters it, control is directed to a user-definable subroutine that can display information relating to the state of the program at that particular point. For example, the contents of important memory locations and of the 65C02 registers can be displayed. If the state is not as expected, then you can start bug-hunting.

Whenever the 65C02 encounters a BRK instruction, the B-flag in the processor status register is set to 1 and then an interrupt sequence much like the one generated by an IRQ signal is started (the main difference is that the

address stored on the stack is the address of the **BRK** instruction plus two). Since the address of the interrupt-handling routine used is the same one used for **IRQ** interrupts, that routine should properly check the status of the **B**-flag to determine what caused the interrupt. In fact, this is what is done on the //c. Once the //c determines that the interrupt was caused by a **BRK** instruction, control is passed to the address stored at the user vector locations **\$3F0** and **\$3F1** (low-order byte first). This vector usually contains **\$FA59**, the address of the subroutine that displays the current contents of all the registers, but can be changed to point to any other interrupt-handling routine that you care to use.

The 65C02 Memory Space on the //c

In this section, we are going to take a look at the layout of the memory space that is available to the 65C02 as implemented on the //c. This memory space can be thought of as being composed of three general parts: **RAM**, **ROM**, and input/output (**I/O**) memory addresses. The //c's main **RAM** memory map is shown in Figure 2-3. Its **ROM** and **I/O** memory map is shown in Figure 2-4.

In the following sections, we will encounter several situations where the same logical memory address is used by more than one actual physical memory location. The //c uses a set of special "soft switches" to select which of these locations is to be active at any given time. (A "soft switch" is a memory location that, when accessed from a software program, causes a change in the //c's hardware environment.) This is necessary because the 65C02 would become hopelessly confused if several locations sharing the same address were active at the same time. We will be looking at the soft switches that the //c uses to manage its memory space in Chapter 8.

RAM Memory

The area of **RAM** memory that is most often used on the //c is that part of "main" (as opposed to "auxiliary") memory that extends from locations **\$0000** to **\$BFFF**. As indicated in Figure 2-3, some regions within this range are dedicated for special uses. Here is a summary of the usage of the main **RAM** memory locations:

- **\$0000-\$00FF**. This is the 65C02 zero page and it is used extensively by all parts of the //c's operating system, including the system monitor (see Chapter 3), the Applesoft interpreter (see Chapter 4), and the disk operating system (see Chapter 5). Those locations available for use by your own programs are shown in Table 2-5.
- **\$0100-\$01FF**. This is the 65C02 stack area and is also used for temporary data storage by the Applesoft interpreter. (See Chapter 4.)

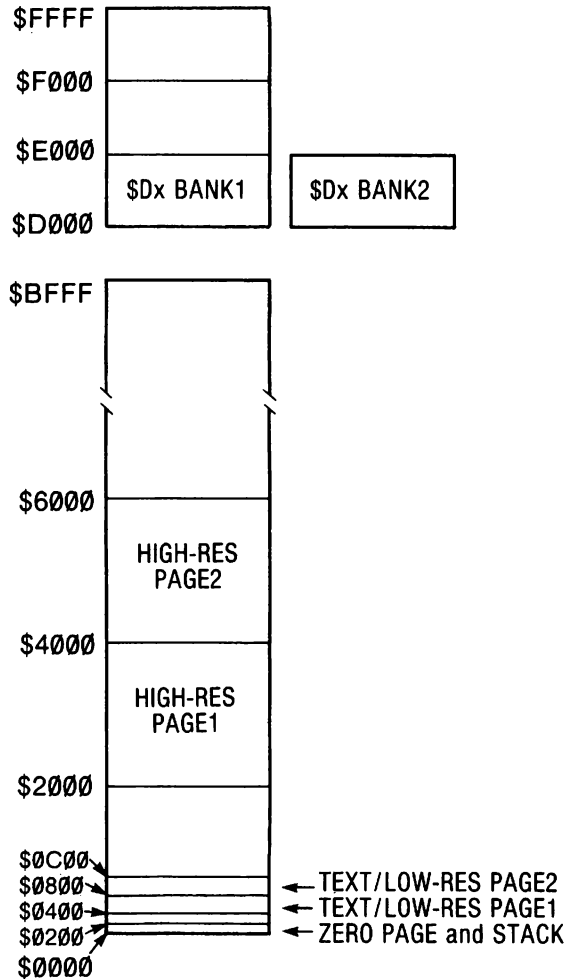


Figure 2-3. Memory map of main RAM.

- **\$0200-\$02FF.** This area of memory is normally used as an input buffer whenever character information is entered from the keyboard or from diskette. (See Chapter 6.)
- **\$0300-\$03CF.** This area of memory is not used by any of the built-in programs in the //c and so is available for use by your own programs. It is an ideal location for storing small assembly-language programs that are called from Applesoft and most of the examples presented in this book are to be loaded here.
- **\$03D0-\$03FF.** Portions of this area of memory are used by the disk operating system, Applesoft, and the system monitor for the purpose of storing position-independent vectors to important subroutines that can

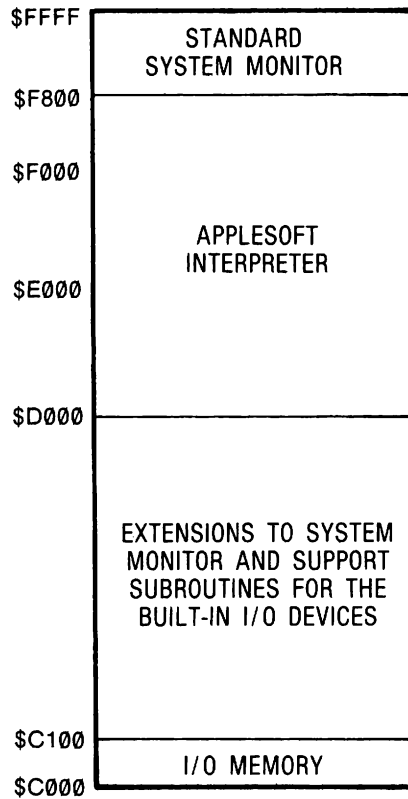


Figure 2-4. Memory map of ROM and I/O memory.

be located anywhere in memory (such as interrupt-handling subroutines). See Appendix IV for a complete description of how this area is used.

- **\$0400-\$07FF.** This is page1 of video memory that is used for displaying both the primary text screen and the primary low-resolution graphics screen. (See Chapter 7.) It is also used for displaying one-half of the text screen when in 80-column mode. Note that there are 64 bytes in this area that are not actually used for the video display; they are called “screen holes” and are used for data storage by the //c’s built-in I/O devices.
- **\$0800-\$0BFF.** This is page2 of video memory that is used for displaying both the secondary text screen and the secondary low-resolution graphics screen. (See Chapter 7.) Since page2 is rarely used, this area of memory is normally used for program storage; in fact, the default starting position for an Applesoft program is \$801.
- **\$0C00-\$1FFF.** This area of memory is free for use.
- **\$2000-\$3FFF.** This is page1 of video memory that is used for displaying the primary high-resolution graphics screen. (See Chapter 7.)

- **\$4000-\$5FFF.** This is page2 of video memory that is used for displaying the secondary high-resolution graphics screen. (See Chapter 7.)
- **\$6000-\$BFFF.** This area of memory is normally free for use. However, the upper part of it (above \$9600) will be used if a disk operating system is installed. (See Chapter 5.)

Main memory also contains an additional 16K of RAM memory that is located from \$D000 to \$FFFF (the 4K block from \$D000 to \$DFFF is duplicated). The ProDOS disk operating system occupies most of this area and so it cannot be safely used by other programs. This 16K area is called bank-switched RAM and will be discussed in detail in Chapter 8.

The //c also comes with 64K of “auxiliary” RAM memory that can be used for program and data storage. This memory occupies the same address spaces as the 64K of main RAM memory and so can be thought of as a “twin” memory space. There are slight differences, however, in how some of the areas within this memory are interpreted. For example, the two memory areas corresponding to the page2 video areas in main memory are not reserved for those purposes in auxiliary memory. Furthermore, the two areas corresponding to page1 video areas are not used for video display purposes unless 80-column text mode is active or unless a double-width graphics mode is active. These differences will be discussed in greater detail in Chapter 7.

Input/Output (I/O) Memory

The //c's I/O memory space corresponds to those addresses from \$C000 to \$C0FF. Although these addresses may be read from or written to in exactly the same way as normal RAM or ROM memory locations, there is no memory stored at these locations. Instead, whenever these locations are accessed, a physical change in the system can be effected (for example, the graphics display can be turned on, the character set can be changed, or the disk drive motor can be turned on), the status of an I/O device can be read, or data can be transferred to or from the I/O device. This method of handling I/O operations is called memory-mapped I/O.

For example, consider the //c's keyboard. The keyboard has been wired into the system in such a way that it can be controlled by using the locations \$C000 and \$C010. (See Chapter 6.) To determine whether a key has been pressed, address \$C000 is examined; if bit 7 at this “location” (the keyboard strobe bit) is 1, then a key has indeed been pressed. Address \$C010 is accessed to clear the keyboard strobe bit. Even though an address is accessed in order to read and clear the keyboard, there is no memory chip on the //c that corresponds to this address.

All of the //c's I/O memory locations will be discussed in later chapters. A summary of the meaning of each of these locations is contained in Appendix III.

Table 2-5. 65C02 zero page locations not used by the system monitor, Applesoft, or ProDOS.

Available Locations:

\$06	\$07	\$08	\$09		
\$19	\$1A	\$1B	\$1C	\$1D	\$1E
\$CE	\$CF				
\$D7					
\$E3					
\$EB	\$EC	\$ED	\$EE	\$EF	
\$FA	\$FB	\$FC	\$FD	\$FE	\$FF

ROM Memory

As you can see from Figure 2-4, ROM memory on the //c extends from locations \$C100 to \$FFFF. Here is a summary of ROM memory usage:

- **\$C100-\$CFFF.** This ROM area contains extensions to the system monitor and subroutines to support the 80-column text display, the printer port, the modem port, the mouse, and the disk drive.
- **\$D000-\$F7FF.** This is the Applesoft ROM space. (See Chapter 4.)
- **\$F800-\$FFFF.** This is the standard system monitor ROM space. (See Chapter 3.)

The permanent programs contained within these ROM areas are often called “firmware” to distinguish them from “software” that is loaded into RAM memory from a diskette.

Note that the addresses used by the Applesoft and system monitor ROMs (\$D000 . . . \$FFFF) are the same as the ones used by the //c’s bank-switched RAM space.

Further Reading for Chapter 2

On 6502 assembly-language programming . . .

MCS6500 Microcomputer Family Programming Manual, MOS Technology, Inc., 1976. This book comes straight from the manufacturer of the original 6502 microprocessor.

R. Zaks, *Programming the 6502*, Sybex, 1978.

L.A. Leventhal, *6502 Assembly Language Programming*, Osborne/McGraw Hill, 1979.

M.L. deJong, *Programming & Interfacing the 6502*, With Experiments, Howard W. Sams & Co., Inc., 1980.

D. Inman and K. Inman, *Apple Machine Language*, Reston Publishing Company, Inc., 1981.

R. Wagner, *Assembly Lines: The Book*, Softalk Books, 1982.

R.C. Haskell, *Apple II 6502 Assembly Language Tutor*, Prentice-Hall Inc., 1983.

On the 65C02 microprocessor . . .

S. Hendrix, "The CMOS 65C02", *Byte*, December 1983, pp. 443–452. This article reviews some of the limitations of the 6502 and introduces the new 65C02 microprocessor.

On machine cycle time . . .

S. Wozniak, "Impossible Dream: Computing e to 116,000 Places With a Personal Computer", *Byte*, June 1981, pp. 392–407. This article has interesting comments on the 6502's effective machine cycle time on the Apple II (it's the same on the //c).

On memory usage by the 6502/Apple II . . .

W.F. Luebbert, *What's Where in the Apple*, Micro Ink, Inc., 1982. This book contains a comprehensive memory map for the Apple II.

Assemblers (software) . . .

The following assemblers operate in ProDOS:

G. Bredon, *Merlin Pro*, Roger Wagner Publishing, Inc., 1984.

G. Bredon, *BIG MAC.C*, A.P.P.L.E., 1984. This assembler is identical to *Merlin Pro* but is available to members of A.P.P.L.E. (Apple Pugetsound Program Library Exchange) only.

ProDOS Assembler Tools, Apple Computer, Inc., 1983.

The following assemblers operate in DOS 3.3:

B. Sander-Cederlof, *S-C Macro Assembler*, S-C Software Corporation, 1983.

R. Hyde, *LISA*, Lazerware, 1981.

G. Bredon, *Merlin Pro*, Roger Wagner Publishing, Inc., 1984.

3

The System Monitor

The system monitor is a machine-language program that resides in the //c's ROM area and whose "cold-start" entry point to a special command interpreter is located at \$FF59. It is called a "monitor" because it supports several commands that allow you to quickly and easily view and modify the contents of memory locations, programs loaded into memory, or 65C02 registers. In addition, commands are available that can be used to run programs, to assist in the debugging of programs, and to perform general housekeeping functions (such as data movement or data comparison).

The subroutines that make up the system monitor take up two large parts of the //c's ROM area. The first part resides from \$F800 to \$FFFF and the second part from \$C100 to \$CFFF. Generally speaking, the first part is comparable to the standard system monitor ROM that resided at the same locations in the earlier Apple //e, Apple II Plus, and Apple II computers; the code is not identical, but virtually all of the starting addresses for its commonly used subroutines are the same as on older models. The internal ROM area from \$C100 to \$CFFF provides the additional space needed for the longer subroutines required to support the //c's 80-column video display. It also holds the support subroutines for the built-in printer, modem, mouse, and disk drive interfaces.

The subroutines contained within the system monitor perform most of the fundamental input/output (I/O) tasks needed to support programs running on the //c. Such tasks include reading a character from the keyboard, displaying a character on the video screen, displaying graphics on the video screen, and reading game controller input. Other subroutines required to support the monitor commands themselves are also found here, of course. In addition, there are numerous utility subroutines used by the code performing these tasks and commands. In the last section of this chapter, we will identify some of the more useful subroutines that can be accessed from Applesoft by using the CALL command or from assembly language by using the JSR (jump-to-subroutine) or JMP (jump) instructions.

The usefulness of the system monitor is greatly enhanced by the fact that, being in ROM, its subroutines and command interpreter are always easily accessible. There are three main entry points to the system monitor command interpreter—OLDRST (\$FF59), MON (\$FF65), and MONZ (\$FF69)—and con-

trol can be passed to them from Applesoft direct mode by entering the commands "CALL -167," "CALL -155," and "CALL -151," respectively. (Note that Applesoft considers a "negative" decimal address to be equivalent to the standard positive address minus 65536; for example, \$FF69 can be represented as 65385 or $65385 - 65536 = -151$.) After this has been done, the system monitor prompt symbol (the asterisk—"*") will appear and you can begin to enter any of the commands that the system monitor supports (or, if ProDOS is active, any valid ProDOS commands).

The //c reacts slightly differently to each of the above three CALLs to its standard entry points. The cold-start entry point, OLDRST (-167), will initialize "normal" video mode (white characters on a black background), select the full-screen text video mode, and then enable the standard keyboard input and video screen output subroutines. It also deactivates the //c's disk operating system (ProDOS) so that it must be reactivated before returning to Applesoft (see the discussion below of the BASIC and CONTINUE BASIC commands). After this has been done, control passes to the primary warm-start entry point, MON (-155), where the 65C02's decimal mode flag is cleared (to force binary arithmetic) and the speaker is beeped. Control then passes to the secondary warm-start entry point, MONZ (-151), which takes care of setting up the "*" prompt symbol and interpreting commands that are entered from the keyboard. MONZ (-151) is the entry point that is most commonly used to enter the system monitor command interpreter.

The System Monitor Commands

The commands that the system monitor supports are summarized in Table 3-1. Before we take a detailed look at these commands, let's review the general command entry rules that must be followed.

First of all, the system monitor "thinks" in hexadecimal. This means that it displays all addresses or data in a standard hexadecimal format and that all information must be provided to it in this format as well. Decimal numbers cannot be used.

Addresses (from \$0000 . . . \$FFFF) must normally be specified as four hexadecimal digits but leading zeros may be omitted if you wish. If an address is entered that is longer than four digits, only the last four digits specified are used. Similarly, byte values (from \$00 . . . \$FF) must normally be specified as two hexadecimal digits but, again, a leading zero may be omitted. If more than two digits are specified for a byte value, only the last two are used.

The DISPLAY Command : Displaying the Contents of Memory

After you have entered the system monitor, you can quickly read and display what is stored in any particular memory location by simply entering the

Table 3-1. Summary of System Monitor Commands.

<i>Command Name</i>	<i>Syntax</i>	<i>Description</i>
DISPLAY	addr1.addr2	Displays the contents of memory from "addr1" to "addr2".
STORE	addr1:b1 b2	Stores the values of bytes "b1", "b2", ... into memory locations beginning at "addr1".
MOVE	addr3<addr1.addr2M	Moves the block of memory from "addr1" to "addr2" to the block beginning at "addr3".
VERIFY	addr3<addr1.addr2V	Compares the block of memory from "addr1" to "addr2" to the block beginning at "addr3" and displays any differences.
EXAMINE	[control-E]	Displays the values to be stored in the 65C02 registers when "G" is used. Follow EXAMINE with STORE to set these values.
GO	addr1G	Runs the program beginning at "addr1".
LIST	addr1L	Disassembles 20 lines of a machine language program beginning at "addr1".
NORMAL	N	Set normal video.
INVERSE	I	Set inverse video.
ADD	b1+b2	Adds the bytes "b1" and "b2" and displays the result.
SUBTRACT	b1-b2	Subtracts byte "b2" from byte "b1" and displays the result.
BASIC	[control-B]	Causes the system to enter Applesoft (cold).

(continued)

Table 3-1. Summary of System Monitor Commands (continued).

Command Name	Syntax	Description
CONTINUE BASIC	[control-C]	Causes the system to enter Applesoft (warm).
USER	[control-Y]	Causes the system to jump to location \$3F8.
KEYBOARD	port[control-K]	Causes the device in "port" to become the source of input.
PRINTER	port[control-P]	Causes the device in "port" to become the current output device.

b1, b2 represent byte values (in hexadecimal)

addr1, addr2, addr3 represent addresses of memory locations (in hexadecimal)

port represents a valid port number (1, 2, 3, 4, 6, 7)

hexadecimal address of the location and pressing [return]. For example, to display the number that has been stored at \$FD0C, you would enter

```
FD0C [return]
```

and the system monitor will respond with

```
FD0C- A4
```

where A4 is the hexadecimal value of the byte stored at \$FD0C. You can also just press [return] by itself to display the contents of the locations immediately after the last one acted on, up to the edge of the next 8-byte boundary (i.e., locations ending in "7" or "F").

The contents of an entire range of memory can be displayed at once by typing in the first address, a period ("."), and then the last address. For example, to examine the 17 bytes of the system monitor ROM area from \$F801 to \$F811, you would enter

```
F801.F811
```

(followed by [return], of course) and you would see the following values displayed (this is called a "hex dump"):

```
F801- 08 20 47 F8 28 A9 0F
F808- 90 02 69 E0 85 2E B1 26
F810- 45 30
```

After the first line, where only those bytes up to the edge of the next 8-byte boundary are displayed, eight bytes will be displayed per line until the very last line where the last few remaining bytes are displayed. The two-digit

values after the dash in each line represent the bytes stored at the address displayed immediately before the dash and in succeeding memory locations.

The STORE Command : Changing the Contents of Memory

It is often handy to be able to quickly enter data into memory locations. You may want to do this in order to provide data to a program, to enter the program itself, or to make quick changes to the program. The system monitor makes this easy by providing you with a convenient command to do this.

To change the contents of memory, you must first type in the address of the first location to be changed, followed by the STORE command (a colon), and then the values of the bytes to be stored in that location and succeeding locations, separated by spaces. For example, to place the values \$3E, \$22, \$24, \$00, and \$29 into addresses \$300 through \$304, you would enter the command:

```
300:3E 22 24 0 29
```

(The number of bytes that can be stored after the colon is limited by the fact that only 255 characters can be entered before pressing [return] or else the line will be cancelled. This allows about 83 data bytes to be specified.)

To continue entering values at this point, you can simply type a colon followed by more data bytes separated by spaces. The address at which the first byte will be stored will automatically be assumed to be the one after the last one that was accessed. Thus, if you entered the command

```
:44 33
```

immediately after entering the above command, address \$305 would contain \$44 and address \$306 would contain \$33.

All of the machine language programs that will be presented in this book can be entered using this technique. To understand how to do this, first refer to Table 3-2, which sets out the assembler source listing of an example program after the assembly process has been completed. This program doesn't do anything really useful, it just prints out all digits from 0 to 9 on the video screen and then stops. What we are really interested in is seeing how to interpret this listing and how it can be used to allow you to enter the program into memory.

First remember that the assembler-listing format used in this book is that used by the Merlin Pro assembler only and that if you are using any other assembler the format may be different. Fortunately, however, formats from one assembler to another are generally quite similar.

The assembler-listing format is made up of six general fields. The first field is the address and data field and can be found at the far left of the listing. Each line in this field contains an address used by the program followed by

Table 3-2. The format of a typical assembly-language program after the assembly process.

Line Number	Address: Data	Label	Instruction	Operand	Comments
1		*****			
2		* EXAMPLE *			
3		*****			
4					Character output subroutine
5		COUT	EQU	\$FDED	
6			ORG	\$300	
7					
8			LDX	#0	
9	0300: A2 00		TXA		Put digit in A
10	0302: 8A 00	DIGITOUT	ORA	#\$B0	Convert to ASCII digit
11	0303: 09 B0		JSR	COUT	
12	0305: 20 ED FD		INX		Go to next digit
13	0308: E8 0A		CPX	#10	Done?
14	0309: E0 0A		BNE	DIGITOUT	No, so loop
15	030B: D0 F5		RTS		
16	030D: 60				

the data byte stored at that address and, in certain cases depending on the type of instruction, at the following one or two addresses as well. This information is all you need to be able to enter the program from the monitor because it is in exactly the same format used by the STORE command. To enter the program, all you must do is enter the following STORE commands:

```
300:A2 0
302:8A
303:9 B0
305:20 ED FD
308:E8
309:E0 A
30B:D0 F5
30D:60
```

Since the program is so short, you could also enter the whole program using just one long STORE command:

```
300:A2 0 8A 9 B0 20 ED FD E8 E0 A D0 F5 60
```

The rest of the fields in the listing simply relate to the source code that gave rise to the machine language bytes that make up the program. These are, in order, the line number field, the label field, the instruction field, the operand field, and the comment field.

A faster way to enter a machine language program that is already stored on diskette is, of course, to use the ProDOS BLOAD command. This is done by entering the command

```
BLOAD FILENAME ,Aaddr
```

where "FILENAME" represents the name of the binary file to be loaded and "addr" represents the decimal starting address at which it is to be loaded, or, if the address is preceded by "\$", the hexadecimal starting address. The ",Aaddr" suffix is optional; if the suffix is omitted, the program will be loaded at the position it was in when it was originally saved to diskette using the BSAVE command.

The MOVE Command : Copying the Contents of Memory

It is sometimes necessary to copy the contents of one block of memory to another part of memory. Two common situations where such a move would be performed are when an assembly-language program is being relocated or when a data block is being duplicated because it may be overwritten by subsequent operations and you don't want to lose it.

You could perform the move by examining the contents of all the memory locations in question and then entering these values at the new locations

using the DISPLAY and STORE commands, but there is an easier way: you can use the MOVE command. The syntax of this command is as follows:

```
{destination}<{sourceS}. {sourceE}M
```

where {destination} represents the address to which the block of memory is to be moved (the destination address), {sourceS} represents the starting address of the block to be moved (the source starting address), and {sourceE} represents the ending address of the block to be moved (the source ending address).

For example, to move the program that you just entered in the previous section, which resides from \$300 through \$30D, to locations \$1000 through \$100D, you would enter the command

```
1000<300.30DM
```

To see that the move has, in fact, been performed, enter the following two commands:

```
300.30D  
1000.100D
```

and compare the two hex dumps. They will be identical apart from the address indicators. (You can also use the VERIFY command to do the comparison automatically; see below.)

When moving a block of memory, you must ensure that the destination address is not within the range of addresses defined by the source block. If it is, then the block will not be properly moved because the area of the source block from the destination location to the end of the block will be overwritten before it is actually moved. This occurs because the byte stored at the lowest-addressed location in the source block is moved first, followed by the rest of the bytes in increasing order of address until the end of the block is reached. For example, if the MOVE command

```
301<300.30DM
```

is entered, the block of memory from \$301 to \$30E will *not* contain an image of \$300 to \$30D before the move but rather will be filled with the value of the byte stored at \$300. You can see why by visualizing the steps that are followed to perform the move: first, the byte at \$300 is moved into \$301, then the byte at \$301 (which has just been overwritten) is moved into \$302, and so on. This type of move is handy for quickly storing the same values at locations throughout an area of memory, but not much else. For example, to zero the area of memory from \$2000 to \$3FFF, you would enter the commands

```
2000:0  
2001<2000.2FFEM
```

One important note on using the MOVE command to relocate machine language programs: many programs will not operate properly at their new locations unless they are modified first. Any program that uses JMP (jump) or JSR (jump-to-subroutine) instructions to transfer control to areas that are

within the block being moved, or that read from or write to addresses within that block, fall within this “unrelocatable” category. This problem arises because such instructions refer to *absolute* memory locations, locations that will not be meaningful after the program has been moved. The easiest way to make a program operate at a new location is to reassemble it at the new location and then enter the new data bytes that the assembler generates. This can be done by changing the operand of the ORG (for “origin”) statement in the assembler source listing (see line 7 of the sample program in Table 3-2) to reflect the new starting address of the program. You could also patch the program manually to fix up all such absolute references in the program by replacing them with the new absolute addresses (low-order byte first), but this is time consuming and prone to error.

The VERIFY Command : Comparing Ranges of Memory

Another useful chore that can be performed by the system monitor is the comparison of the contents of two blocks of memory. Comparisons are commonly made for the purposes of determining the locations at which two similar programs (usually related revisions) differ from one another.

You could perform the comparison manually by repeatedly using the DISPLAY command but this would be tedious at best, especially for long data blocks. The process can be automated, however, by using the VERIFY command. The syntax for this command is as follows:

```
{block2}<{blockS}. {blockE}V
```

where {block2} represents the starting address of the block of memory to which comparisons will be made, {blockS} represents the starting location of the main block, and {blockE} represents the ending location of the main block. When the command begins to execute, each byte in the main block will be compared with its corresponding byte in the other block. If there are any differences, then they will be printed out in the following format:

```
{address} - 34 (EA)
```

where {address} is the address of the byte in the main block that is different, the first (unbracketed) data byte represents the value of that byte in the main block and the second number represents the value of that byte in the other block.

The EXAMINE Command : Examining the 65C02's Registers

The system monitor reserves several locations in zero page for temporary storage of the 65C02's internal registers, A, X, Y, P, and S. All of these registers

(except for the stack pointer, S) are loaded with the values stored at these locations whenever the monitor's GO command is entered (see below). This allows you to properly initialize the 65C02 registers before executing any assembly-language program.

The saved contents of the 65C02's internal registers can be examined at any time by using the EXAMINE command by entering the following control character:

`[control-E]`

(Recall that this notation means "press the CONTROL key and, while it is being held down, press the E key.") When the EXAMINE command is entered, the currently saved values of each of the five 65C02 registers will be displayed in the following typical format:

`A=02 X=CC Y=D8 P=00 S=B7`

In this list, A represents the accumulator, X and Y represent the X and Y index registers, P represents the processor status register, and S represents the stack pointer. The two-digit hexadecimal number after each "equal" sign indicates the current value of the corresponding register.

Immediately after the [control-E] command has been entered and the contents of the registers have been displayed, you can set any of the register locations to any value that you want by entering a colon followed by the new values for the contents of the registers, separated by spaces. The new values must be entered in the order in which the registers are displayed. If you want to change some, but not all, of the registers, then you will have to enter the current values for those of the other registers that are displayed before the last one that you wish to change.

For example, if you want to set the X register to \$33 and leave the other registers unchanged, you would enter the command

`:02 33`

where 02 represents the current value of the accumulator.

The [control-E] command is primarily used as a debugging tool when developing an assembly-language program. Program subroutines that require certain registers to be initialized in certain ways before they will perform properly can easily be tested by setting up the registers after entering [control-E] and then executing the subroutine.

The GO Command : Running a Program

You can run any machine language program that is contained in memory by using the monitor's GO command. To do this, you must type in the starting address of the program followed by "G" and then press [return]. Before

control is passed to the program, the 65C02's A, X, Y, and P registers are loaded with the values last set by the EXAMINE command (see above). When the program stops running, you will usually return to the system monitor command interpreter and see the "*" prompt symbol once again.

For example, if you want to run a program that starts at location \$300, then you would enter the command

```
300G
```

The LIST Command : Disassembling Assembly-Language Programs

The LIST command can be used to translate bytes in any area of memory into the assembly-language mnemonics they represent and to display the listing on the screen. This command essentially reverses the process performed by an assembler and so the function it performs is called "disassembly."

A disassembled listing of memory is much more comprehensible and informative to a programmer than a simple hex dump that only displays raw numbers. It is especially useful as an aid in debugging assembly-language programs that have been loaded into memory. The syntax associated with the LIST command is as follows:

```
{address}L
```

where {address} represents the address at which you want to begin the listing. A total of twenty disassembled lines will be displayed for each "L" specified after the address.

Let's examine an area of the //c's system monitor ROM to observe the format in which the LIST command generates its output. As we will see later, the basic character input routine used by the Monitor begins at location \$FD0C and is called RDKEY. To disassemble the RDKEY subroutine, enter the command

```
FD0CL
```

and you will see the following 20-line display:

FD0C-	A4 24	LDY	\$24
FD0E-	B1 28	LDA	(\$28),Y
FD10-	EA	NOP	
FD11-	EA	NOP	
FD12-	EA	NOP	
FD13-	EA	NOP	
FD14-	EA	NOP	
FD15-	EA	NOP	

FD16-	EA			NOP	
FD17-	EA			NOP	
FD18-	6C	38	00	JMP	(\$0038)
FD1B-	91	28		STA	(\$28),Y
FD1D-	20	4C	CC	JSR	\$CC4C
FD20-	20	70	CC	JSR	\$CC70
FD23-	10	FB		BPL	\$FD20
FD25-	48			PHA	
FD26-	A9	08		LDA	#\$08
FD28-	2C	FB	04	BIT	\$04FB
FD2B-	D0	1D		BNE	\$FD4A
FD2D-	68			PLA	

Each line in this listing represents a starting address, the machine language bytes representing the 65C02 instruction opcode and its operand, the three-letter mnemonic for the instruction, and the formatted operand. Note that operands that have a "\$" prefix represent an address and that those that have a "\$#" prefix represent immediate hexadecimal data. In addition, the operand after any branch instruction (BEQ, BNE, BPL, and so on) is the absolute address of the "branched-to" location rather than the relative address of that location. The 65C02 uses relative addresses only, but it is the absolute address that is usually more meaningful because it allows a programmer to more easily follow the flow of the program.

Note that you can continue to disassemble twenty more lines beginning at the address immediately after the last disassembled byte by entering the "L" command without an address. Multiple "L"s can also be entered to disassemble more than twenty lines at once; for example, "LLLL" allows you to disassemble eighty consecutive lines.

When you are disassembling an area of memory you may sometimes see a "???" indicator in the opcode field instead of a standard 65C02 mnemonic. The system monitor's disassembler subroutine uses this triad of question marks whenever it is unable to convert the contents of memory into a valid 65C02 instruction. This might happen if you are attempting to disassemble an area of memory that contains program data or encoded text rather than instructions or if you begin disassembling in the "middle" of an instruction (remember that 65C02 instructions can be up to three bytes long). If you suspect that you have started in the middle of an instruction, try disassembling from a location that is one or two locations away from the original starting location.

In many cases, a data area will erroneously be interpreted as a series of valid instructions by the disassembler. For example, a zeroed out data area would appear as a series of BRK instructions. This is because the machine language byte for BRK is 00. Such data areas are usually obvious, however, because the "program" they appear to define is clearly meaningless or out of context.

The NORMAL and INVERSE Commands : Changing Video Display Modes

Monitor operations that affect the video display can be performed either in normal video (white characters on a black background) or in inverse video (black characters on a white background). To select the inverse video format, enter the command

```
I [return]
```

To select the normal video format, enter the command

```
N [return]
```

You will probably not have to use these commands very often.

The ADD and SUBTRACT Commands : Simple Arithmetic

You can perform simple one-byte hexadecimal arithmetic while in the system monitor by taking advantage of its ADD and SUBTRACT commands. To add two numbers together, you would enter the command

```
{number1}+{number2}
```

where {number1} and {number2} represent the two one-byte hexadecimal numbers to be added. The result of the addition will be shown on the next video display line.

The subtraction command is similar. To subtract one number, say {number2}, from another, say {number1}, you would enter the command

```
{number1}-{number2}
```

and the result will be calculated and displayed.

The result that either the ADD or SUBTRACT command displays is a one-byte number only. This means that any overflow or underflow in the arithmetic calculation is ignored.

The BASIC and CONTINUE BASIC Commands : Entering Applesoft

The system monitor supports two commands that can be used to transfer control from the monitor to Applesoft direct mode (as indicated by the "]" prompt symbol). These are the [control-B] and [control-C] commands. There are also subroutines that can be called to enter Applesoft that begin at \$0000 (with or without ProDOS) and \$03D0 and \$3D3 (only when ProDOS is being used).

The BASIC command, [control-B], is used to re-enter Applesoft in such a way as to cause it to be reinitialized. This is called a “cold start” and will cause any Applesoft program which may be residing in memory to be removed.

The CONTINUE BASIC command, [control-C], is used to re-enter Applesoft in such a way that the existing Applesoft program and the values of its variables are not affected at all. This is called a “warm start.” An alternative way to warm-start Applesoft is to call a subroutine that begins at \$0000 by entering the command “0G”.

The effect on the ProDOS disk operating system must also be considered when moving to Applesoft from the monitor. If you are using ProDOS and the monitor was entered with either a CALL -151 or CALL -155 command (the warm-start entry points), then ProDOS will still be active upon the return to Applesoft using [control-C]. The [control-B] command, however, will cause a NO BUFFERS AVAILABLE error message to be displayed whenever a ProDOS I/O command is attempted. This renders ProDOS useless and so you should never use [control-B] to return to Applesoft when ProDOS is active.

If the monitor was entered via its cold-start entry point with a CALL -167 command, ProDOS will be deactivated after a [control-B] and [control-C] command is entered to cause a return to Applesoft. In this situation, ProDOS can be reactivated by entering a CALL 976 command, but this causes the values of any active Applesoft program variables to be cleared. Note, however, that even after the CALL 976 is entered, ProDOS will still be rendered unusable if it was entered with a [control-B] command for the reasons given in the previous paragraph.

Applesoft can always be entered with ProDOS active by using a “3D0G” command (\$3D0 is the address of a subroutine that performs a warm start of ProDOS), but this method is not recommended because of zero page memory conflicts between ProDOS and the system monitor and the fact that any active Applesoft variables will be cleared.

In summary, to ensure that you never deactivate ProDOS or clear the values of any active Applesoft program variables, you should always enter the monitor at one of its two warm-start entry points (-151 or -155) and always return to BASIC using the [control-C] command.

The USER Command : User-Defined Command

The system monitor is flexible enough to allow you to define the actions to be taken whenever its special USER command, [control-Y], is entered. The [control-Y] command causes the monitor to perform an unconditional jump to location \$3F8. By placing a 65C02 JMP instruction there (which behaves like an Applesoft GOTO), followed by the two-byte address (low byte first) of the start of the subroutine that you want to execute, you can easily make the [control-Y] command execute any program you wish.

Let's take a look at a simple example of how to take advantage of the USER

command. The first thing you have to decide is what you want to happen when [control-Y] is pressed—that's easy. Then you must write the program to perform what it is you want to do—not so easy. We can, however, make use of subroutines that already exist in the //c's ROM areas to perform many useful chores. For example, there is a subroutine beginning at \$FC58 that can be called to clear the video screen and a subroutine beginning at \$FD0C to read a key from the keyboard. To set things up so that when the USER command is entered, the system pauses until a key is pressed and then clears the screen, a "JMP \$0300" instruction must be set up at \$3F8 and then "JSR \$FD0C" and "JMP \$FC58" instructions must be stored beginning at \$300. This can be done by using two STORE commands as follows:

```
3F8:4C 00 03
```

("4C" is the opcode for the JMP instruction and "00 03" is the address of the user-defined subroutine—low-order byte first) and

```
300:20 0C FD 4C 58 FC
```

where "20 0C FD" are the data bytes for "JSR \$FD0C" (\$20 is the opcode for the JSR instruction) and "4C 58 FC" are the data bytes for "JMP \$FC58". Now when you enter [control-Y] the //c will wait until you press a key and then the screen will be cleared!

Note that you cannot simply place the entire subroutine at \$3F8, because only locations \$3F8 to \$3FA are reserved for use by the USER command. Locations after that are reserved for other purposes and must not be overwritten.

Parameters can be passed to the USER command by storing them in memory just before the monitor executes the USER command. This can be done by using the STORE command. If the parameters to be passed represent addresses, there is a much more convenient way to pass up to three of them. For example, if the USER command is entered as follows:

```
addr1<addr2.addr3[control-Y]
```

then "addr1" will be stored at monitor locations A4L (\$42) and A4H (\$43), "addr2" will be stored at A1L (\$3C) and A1H (\$3D), and "addr3" will be stored at A2L (\$3E) and A2H (\$3F). Each of these addresses is stored with its lower two digits in the first of the two memory locations specified for each parameter. Two addresses can be passed (in A1L/A1H and A2L/A2H) by removing the "addr1<" part in the above command line and one address can be passed (in A1L/A1H) by removing the "addr1<addr2." part.

The KEYBOARD and PRINTER Commands : Redirecting Input and Output

The system monitor provides two simple commands that allow you to easily redirect the source of character input and output to one of the //c's built-in I/O

ports (the printer, modem, 80-column display, mouse, or disk drive). These are the **KEYBOARD**, [control-K], and **PRINTER**, [control-P], commands, respectively. They perform exactly same the functions as Applesoft's **IN#** and **PR#** commands.

The syntax associated with both of these commands is similar:

```
{port number}[control-K]
```

for the **KEYBOARD** command and

```
{port number}[control-P]
```

for the **PRINTER** command, where {port number} is a digit which can take on the values 1 (printer port), 2 (modem port), 3 (80-column video port), 4 (mouse port), 6 (internal disk drive port), or 7 (external disk drive port) and represents the port number of the device to which you wish to pass control. You can also specify a slot number of 0; if you do this when entering the **KEYBOARD** command, the keyboard will become the source of character information. If you do this when entering the **PRINTER** command, the video screen will become the current output device.

The **KEYBOARD** command is usually used to "connect" alternate input devices such as an external keyboard or a modem to the //c by vectoring all requests for input to them. The **PRINTER** command is usually used to activate a printer so that you can obtain a hardcopy printout of your activities while in the monitor. To turn on a printer that is connected to port 1 (the //c's standard printer port) you would enter the command

```
1[control-P]
```

After this is done, all outputted characters will be sent to the printer instead of the video screen.

Another common use for the **PRINTER** command is to "boot" the disk drive. This can be done by entering the command:

```
6[control-P]
```

Note that whenever the **KEYBOARD** or **PRINTER** command is entered, the monitor jumps to location \$Cs00 (where "s" is the port number specified), which is the first address of a driver program for the particular port in question. It is the driver program in ROM that dictates exactly how the I/O is to be redirected.

I/O is redirected on the //c by changing the addresses stored in two vectors in zero page, the input link and the output link. The use of these links will be discussed in detail in Chapters 6 and 7.

Note that because of the way ProDOS operates, the **KEYBOARD** and **PRINTER** commands may not work properly in a ProDOS environment. This is because ProDOS is forever storing the addresses of its own input and output subroutines in the I/O links; as soon as this is done, the new input or output

device is disconnected. Methods of avoiding these problems will also be discussed in Chapters 6 and 7. In summary, if ProDOS is active, then use its PR# and IN# commands while in the system monitor and not the monitor's KEYBOARD and PRINTER commands.

Multiple Commands on One Line

All of the examples that we have given so far have contained only one monitor command per line. The monitor is not fussy about this, however, and you can actually put as many commands on one line as that line can hold (a line must be less than 256 characters long).

There are a few syntactical rules to follow, however. First of all, each command on the line must be separated from the next one by a space unless both adjacent commands are one of the letter commands (L, G, M, V, I, N), in which case they can be jammed together.

Second, any command that immediately follows the data bytes after the STORE command must be a letter command without a preceding address. A convenient command to use for this purpose is the NORMAL command ("N") since it is really a "do-nothing" letter command.

Let's look at a few examples of multiple command entry to see how it works.

1. **300LLL** will disassemble 60 lines of a program beginning at \$300 at once.
2. **300:4C 3A FF N 300G** will enter a short program beginning at \$300 to beep the speaker and then execute it (note the "N" after the data bytes of the STORE command).
3. **300.320 800.830** will display two separate blocks of memory, \$300 . . . \$320 and \$800 . . . \$830 one after the other.
4. **3F8:4C 00 03 N 300:4C 58 FC N [control-Y]** will set up the USER command jump address, enter the program to be jumped to, and then execute the USER command (which causes the screen to clear in this example).

System Monitor Subroutines

As we have already seen, the system monitor is made up of several useful subroutines. Most of these subroutines can easily be accessed from Applesoft or assembly-language programs.

Direct access from Applesoft is achieved by using the Applesoft CALL command. Note, however, that only those monitor subroutines that require no initialization of the 65C02 registers can be CALLED in this way because there are no Applesoft commands available to you to set up these registers directly.



One way to access subroutines that require register initialization would be to CALL a RAM-based program that would set up these registers explicitly and then call the requested subroutine. An alternative method makes use of the monitor's GO command and the fact that GO initializes the 65C02's registers to the values stored in zero page by the EXAMINE command before control is passed to the subroutine whose address is stored at \$3A and \$3B (low-order byte first). The values of the registers A, X, Y, and P are stored at locations \$45, \$46, \$47, and \$48, respectively. To execute the subroutine, you must first use the Applesoft POKE command to store the address of the subroutine to be executed at \$3A/\$3B and to store the appropriate register values at locations \$45–\$48. The final step is to execute the GO command by entering it at the point where it sets up the registers before passing control to the address at \$3A/\$3B. This is location \$FEB9 (65209).

For example, you can set up a simple decimal-to-hexadecimal conversion program from Applesoft by calling a monitor subroutine called PRINTYX (\$F940). This subroutine prints out the Y and X registers as four hexadecimal digits (the two most-significant digits are held in Y). To get the converter to work, all you have to do is take your decimal number, divide it by 256, and put the quotient in Y (this represents the decimal value of the two high-order digits) and the remainder in X (this represents the decimal value of the two low-order digits). Here is an example of such a program:

```

100 DEF FN MD(Z) = Z - 256 * INT(Z / 256)
110 INPUT "ENTER A NUMBER: ";N
120 ADDR = 63808 : REM ADDRESS OF "PRINTYX"
    ($F940)
130 POKE 70, FN MD(N):REM SET UP "X"
140 POKE 71, INT (N/256):REM SET UP "Y"
150 POKE 58, FN MD(ADDR) : REM SET UP ADDR LOW
160 POKE 59, INT (ADDR/256) : REM SET UP
    ADDR HIGH
170 CALL 65209    REM CALL "GO" AT $FEB9

```

Line 100 in this program defines a “modulo 256” function that can be used to calculate the decimal value of the lower two digits of a hexadecimal number (0 . . . 255).

These complications do not really arise when calling monitor subroutines from an assembly-language program because the 65C02 has explicit commands for initializing registers (LDA, LDX, LDY, and so on). Once the registers have been properly set up, you can execute the subroutine by using a JSR instruction (like an Applesoft GOSUB) or a JMP instruction (like an Applesoft GOTO).

Some of the more useful subroutines available in the system monitor are set out in Table 3-3. These subroutines are presented in increasing order of address and a symbolic name for each address is shown immediately after the address.

Table 3-3. Apple //c system monitor subroutines.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$F940	(63808)	PRINTYX	Prints out the number held in X (low) and Y (high) as four hexadecimal digits.
\$FB1E	(64286)	PREAD	Reads the current value of the game controller input. On entry, X = game controller number (0 or 1). On exit, Y = game controller reading (0 . . . 255) and A is destroyed.
\$FBC1	(64449)	BASCALC	Calculates the address of the first location used by the current video line. On entry, A = video line number (0 . . . 23). On exit, the address is stored in BASL (\$28) and BASH (\$29), low byte first, and A is destroyed.
\$FC22	(64546)	VTAB	Moves the cursor to the video display line indicated by CV (\$25). On entry, CV must contain the line number required (0 . . . 23). On exit, the base address for the line is set up in BASL (\$28) and BASH (\$29) and A is destroyed.
\$FC42	(64578)	CLREOP	Clears the screen display from the current cursor position to the end of the screen without changing the position of the cursor. On exit, A and Y are destroyed.
\$FC58	(64600)	HOME	Clears the screen display and positions the cursor at the left of the first line on the screen. On exit, A and Y are destroyed.
\$FC62	(64610)	CR	Moves the cursor to the first position of the next video display line (and scrolls if required). On exit, A and Y are destroyed.
\$FC9C	(64668)	CLREOL	Clears the screen display from the current cursor position to the end of the line without changing the cursor position. On exit, A and Y are destroyed.

(continued)

Table 3-3. Apple //c system monitor subroutines (continued).

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$FCA8	(64680)	WAIT	Causes a delay of $0.5 * (26 + 27 * A + 5 * A * A)$ microse-conds. On exit, A is destroyed.
\$FD0C	(64780)	RDKEY	Receives a character of information from the currently active input device (the address for the input subroutine for this device is held in KSWL (\$38) and KSWH (\$39)). On exit, A contains the inputted character and Y is destroyed; other registers may be destroyed, depending on the input subroutine for the input device.
\$FD1B	(64795)	KEYIN	Receives a character of information from the keyboard. On exit, A contains the inputted character and Y is destroyed.
\$FD35	(64821)	RDCHAR	Receives a character of information from the currently active input device and enables escape sequences. On exit, A contains the inputted character and Y is destroyed; other registers may be destroyed, depending on the input subroutine for the input device.
\$FD6A	(64874)	GETLN	Receives a line of information (terminated by [return]) from the currently active input device and places it into the input buffer at \$200 . . . \$2FF. On entry, the prompt symbol to be used must be stored in PROMPT (\$33). On exit, the line is stored in the input buffer beginning at \$200, X contains the number of characters in the line, and A and Y are destroyed.
\$FD6A	(64986)	PRBYTE	Displays a byte as two hexadecimal digits. On entry, A contains the byte to be displayed. On exit, A is destroyed.

(continued)

Table 3-3. Apple //c system monitor subroutines (continued).

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$FDED	(65005)	COUT	Sends a character of information to the currently active output device (the address for the output subroutine for this device is held in CSWL (\$36) and CSWH (\$37)). On entry, A contains the byte to be sent. On exit, registers may be destroyed, depending on the output subroutine for the output device.
\$FDF0	(65008)	COUT1	Displays a character of information on the video display screen at the current cursor position. The display mode is set by logically ANDing the byte with INVFLG (\$32). On entry, A contains the byte to be displayed (with its high bit set to one). On exit, all registers are preserved.
\$FF69	(65385)	MONZ	Enters the //c's system monitor. On exit, all registers are destroyed.

Table 3-3 by no means represents a complete list of the monitor's subroutines. To examine all the subroutines for yourself, you should consult Apple's published source listing of the monitor ROM in "The Apple //c Reference Manual", Volume 2.

Further Reading for Chapter 3

On system monitor subroutines . . .

The Apple //c Reference Manual, Volume 2, Apple Computer, Inc., 1984.

This manual contains the source code for the system monitor on the //c.

W.E. Dougherty, The Apple II Monitors Peeled, Apple Computer, Inc., 1981.

A detailed look at the system monitors for the Apple II and Apple II Plus.

4

Applesoft BASIC

Applesoft BASIC is a high-level programming language interpreter that occupies 10K of the //c's ROM space from location \$D000 through location \$F7FF. (BASIC is an acronym for Beginner's All-Purpose Symbolic Instruction Code.) It is yet another version of the "basic" BASIC developed by Microsoft Corporation of Bellevue, Washington, and so is structurally similar to Microsoft-developed BASICs running on many other personal computers, including those manufactured by Tandy/Radio Shack, Commodore, and IBM.

The //c version of Applesoft is slightly different from the one used on the Apple //e, Apple II Plus, and Apple II. The major changes that have been made are as follows:

- The Applesoft cassette tape commands are accepted but are treated just like the & (ampersand) command. These are the SHLOAD, RECALL, STORE, LOAD, and SAVE commands.
- Applesoft commands can be entered in upper- or lowercase.
- Program lines are listed on the screen beginning in column two in order to facilitate screen editing. (When you move the cursor up to edit the line, it is now positioned over the first digit in the line number.)
- The low-resolution graphics commands have been modified in order to support the //c's special double-width low-resolution graphics display mode. (See Chapter 7.)

Fortunately, none of these changes should affect the performance of the vast majority of programs written for the previous version of Applesoft.

What exactly is the Applesoft programming language, anyway? Well, it's really just another 65C02 assembly-language program, but one that has a special goal: to allow you to easily write your own programs using straightforward, English-like commands. These commands can be used in such a way as to allow you to manipulate various types of data and to perform input/output functions. In addition, Applesoft comes with a built-in editing environment that facilitates creation of its programs.

Applesoft is actually a language "interpreter" and a program is simply a set of data that the Applesoft code in ROM is continuously analyzing (interpreting) to determine what commands are to be executed and in what order.

Other types of BASICs, called “compilers,” are also available. Compilers are simply preprocessors that convert your program source code into directly executable machine language that can then be run just like any other machine-language program. Since directly executable code is generated, no interpretation is necessary when the code is actually executed (except, of course, by the microprocessor) and so the program will run much faster than its interpreted counterpart. Although Applesoft compilers are available, none have been officially released by Apple itself.

The purpose of this chapter is *not* to teach you how to program in the Applesoft language. In fact, you will be presumed to be familiar with Applesoft already. What we are going to do is take a close look at the internals of Applesoft to see how the interpreter performs its various duties. This will include a look at how an Applesoft program and its variables are stored and arranged in memory and how the program is actually executed by the Applesoft interpreter. We will also take a look at how Applesoft can be linked to machine-language subroutines to improve program speed and efficiency.

The study of the internal structure of Applesoft is difficult and frustrating because no official source listing for its code has been made available by Apple. Such a study is not totally futile, however, because it is possible to disassemble the contents of the Applesoft ROM (using the monitor’s “L” command or some other disassembler) to view the language in a convenient assembler-language form that can sometimes be made intelligible (if you’re lucky). In addition, at least two “unofficial” source listings of Applesoft have been published (see the references at the end of this chapter).

Knowledge of the internal structure of Applesoft is important for three main reasons. First, by analyzing the work of the professional programmers who wrote the language you might develop better personal programming practices. Second, you can generally write much more elegant and efficient assembly-language routines to be used in conjunction with Applesoft programs if the routines use the standard routines found in Applesoft because this spares you from having to redevelop the same code from scratch. Third, it is possible to write much more efficient Applesoft programs if you understand how they are being executed.

Applesoft Memory Map

The Applesoft interpreter in ROM uses most of the RAM space located from \$0000 to \$95FF in the main memory area of the //c for program and variable storage and for work areas. (The //c’s other memory area, “auxiliary” memory, will be discussed in Chapter 8.) The area of RAM memory above this, from \$9600 to \$BFFF, is reserved for use by BASIC.SYSTEM (a ProDOS/Applesoft interface program) and the ProDOS disk operating system itself. (See Chapter 5). As we will see in Chapter 5, ProDOS actually “steals” space from Applesoft

when disk files are opened, but this is done in such a way that the operation of the Applesoft program is not disturbed.

Much of the 65C02 zero page (\$0000 . . . \$00FF) is used by Applesoft to hold short subroutines, temporary data areas, and several two-byte pointers that contain the addresses of important data areas used by the program. For example, there are pointers that hold the starting and ending addresses of the program itself, of the space reserved for simple variables and array variables, and of the space reserved for string data. We'll be looking at these pointers in greater detail later on in this section.

(To review, a pointer is a pair of bytes that are positioned in adjacent memory locations and that contain the base address of an area in memory to which they are said to be pointing. The lower half of this address is stored in the byte that is lower in memory. To calculate the absolute address of the area being pointed to, take the number held in the first location and add it to 256 times the number in the second location.)

Page one of memory (\$100 . . . \$1FF) is implicitly used by Applesoft since the 65C02 microprocessor uses this page as its stack. In addition, Applesoft uses the stack area for temporary storage of information when it executes instructions such as FOR/NEXT, GOSUB/RETURN, and ONERR GOTO that need space to hold transfer-of-control information and when it converts binary numbers into decimal numbers.

Applesoft uses page two of memory (\$200 . . . \$2FF) as its character input buffer. For example, whenever an Applesoft program executes the INPUT command to read a line from the keyboard, it initially stores the response in this buffer and then processes it and moves it up into a space reserved for string data near the end of the RAM space reserved for use by Applesoft.

The lower part of page three of memory from \$300 . . . \$3CF is not used by Applesoft and so is a good place to store short assembly-language programs or other data. However, the entire upper part of this page, from \$3D0 . . . \$3FF, is reserved for use by ProDOS, the system monitor (to hold the USER vector and the 65C02 RESET, IRQ, NMI, and BRK interrupt vectors), and by Applesoft. Applesoft reserves the three bytes beginning with \$3F5 for use with its & (ampersand) command. Thus, the upper part of page three should not be overwritten unless it is for the specific purpose of modifying the information stored there. Appendix IV contains a complete memory map of the area in page three from \$3D0 . . . \$3FF.

Pages four through seven (\$400 . . . \$7FF) are used for the //c's primary text display screen. (A secondary text display screen can also be enabled that uses pages eight through eleven (\$800 . . . \$BFF), but it is rarely used.) See Chapter 7 for more information on how the //c interprets these pages.

The rest of the RAM space, from \$800 up to \$95FF, is usually available for storage of the Applesoft program itself and of any variables that it may use. Figure 4-1 shows a generalized Applesoft memory map that indicates the

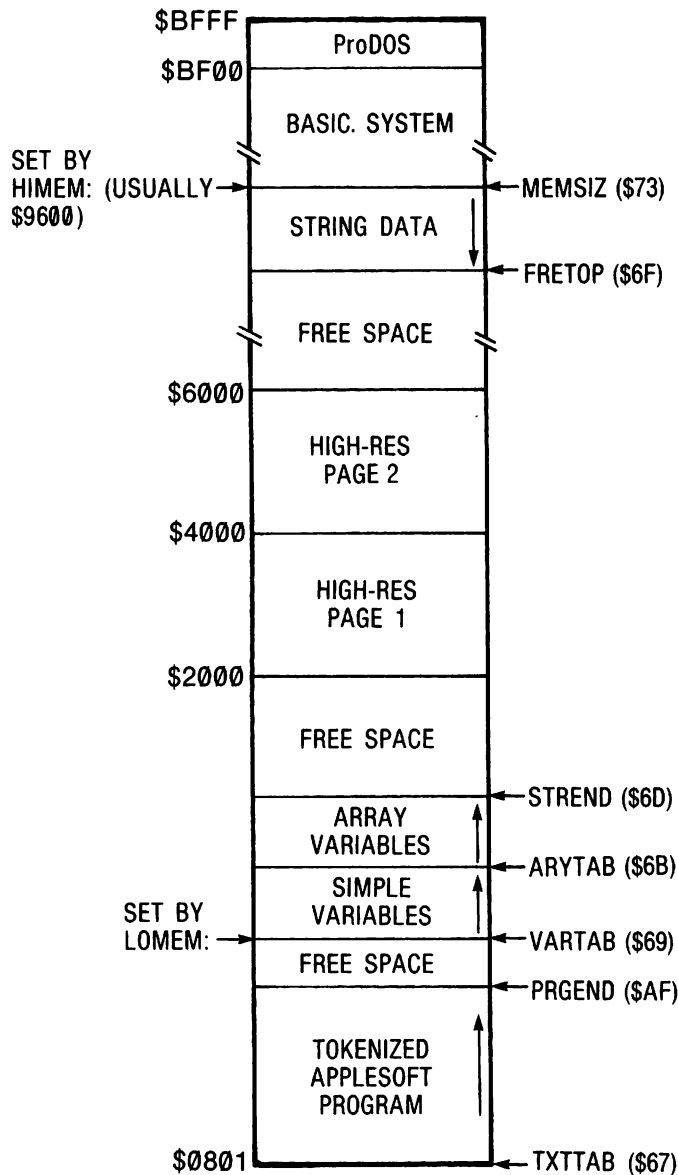


Figure 4-1. Applesoft memory map and data pointers.

relative positions of the program and its variable spaces. The pointers to these areas are all held in zero page and are summarized in Table 4-1.

The Applesoft program itself is usually stored beginning at location \$801, which is the default value of TXTTAB (\$67), the start-of-program pointer. The byte stored at the location immediately before this location (usually \$800) must always be zero. The space used to store information relating to program variables usually starts immediately after the end of the program at the

Table 4-1. Applesoft pointer locations.

<i>Pointer Hex</i>	<i>Location (Dec)</i>	<i>Symbolic Name</i>		<i>Description</i>
\$67	(103)	TXTTAB	(low)	Start of Applesoft program (normally \$801).
\$68	(104)		(high)	
\$69	(105)	VARTAB	(low)	Start of simple variable space. This space usually begins right after the end of the program. However, it can be set higher by using the Applesoft LOMEM: command.
\$6A	(106)		(high)	
\$6B	(107)	ARYTAB	(low)	Start of array space. This space begins right after the end of simple variable space.
\$6C	(108)		(high)	
\$6D	(109)	STREND	(low)	End of variable space.
\$6E	(110)		(high)	
\$6F	(111)	FRETOP	(low)	Start of string space. Applesoft strings are stored from here to just before the address pointed to by MEMSIZ (\$73).
\$70	(112)		(high)	
\$73	(115)	MEMSIZ	(low)	End of string space plus 1 and last location available to Applesoft plus 1. Applesoft strings are stored from this locations down to FRETOP (\$6F). This location is usually \$9600 (when using ProDOS) but can be set lower by using the Applesoft HIMEM: command.
\$74	(116)		(high)	
\$AF	(175)	PRGEND	(low)	End of Applesoft program plus 1 or 2. The end of an Applesoft program is signified by three consecutive "0" bytes. The first "0" is the end-of-line marker for the last line in the program and the next two "0"'s are the "address" of the next line.
\$B0	(176)		(high)	

location pointed to by VARTAB (\$69), the start-of-simple-variables pointer. The position of the start of variable space, however, can be selected by using the Applesoft LOMEM: command before any variables have been defined in the program. This allows you to create a free space between the end of the program and the beginning of the variables that will not be overwritten and that could be used to hold, for example, a machine-language subroutine that is called by the Applesoft program.

Applesoft supports two fundamental classes of variables: array variables and simple variables. Array variables can hold real numbers, integer numbers, or strings; simple variables can hold any of these three types of variables and a special function variable as well (more on this later). An array variable is one that is a member of a collection of variables that are referred to by the same name but that are distinguished from one another by specifying a subscript for each dimension of the array. For example, the variable `AB(3,4,2)` is the “3,4,2” element of a three-dimensional array called “AB”. A simple variable is simply one that is not an element of such an array and that is specified by name only and not by a subscript.

Applesoft keeps information relating to simple variables in a contiguous block of memory that begins at the address pointed to by `VARTAB` (\$69) and ends at the address just before the one pointed to by `ARYTAB` (\$6B). Information relating to array variables begins at the address pointed to by `ARYTAB` and ends at the address pointed to by `STREND` (\$6D).

After the end of the array variable space comes a free space that ends at the address pointed to by `FRETOP` (\$6F), the start-of-string-space pointer. Generally speaking, the contents of string variables are stored from here to the highest available location in memory (usually \$95FF). The `MEMSIZ` (\$73) pointer contains this address plus 1. Strings grow down in memory, so that when more strings are defined, they are placed in memory just below the value contained in `FRETOP` and then `FRETOP` is reduced by the length of the string.

The value of `MEMSIZ` can be lowered by using the Applesoft `HIMEM:` command. This is usually done to provide a safe area for the storage of machine-language programs (we’ll present the details of how to do this in Chapter 5), but it is also commonly done to avoid storing variable data within either of the //c’s two 8192-byte high-resolution graphics screen areas (if this happens, the data will likely be destroyed when a graphics command is executed). These two areas are located from \$2000 . . . \$3FFF and from \$4000 . . . \$5FFF. For example, to “protect” the first high-resolution graphics screen, you would enter the command `HIMEM:7168`. This sets `MEMSIZ` to 1024 bytes below the start of the graphics screen memory area (\$2000–\$400 is equal to 7168 decimal). You can’t just set `HIMEM:` to 8192 (\$2000) because ProDOS uses the 1024 bytes above `HIMEM:` as a general purpose file buffer. (See Chapter 5.)

Note that the free space between the end of the array variables and the beginning of the string data will become smaller and smaller as more variables are defined and as more strings are defined. When all of the free space has been used up, an `OUT OF MEMORY` error message will be generated.

In the next few sections, we will discuss the data spaces used by Applesoft in greater detail.

Tokenization of Applesoft Programs

An Applesoft program is simply the data the Applesoft interpreter acts on in order to determine exactly what instructions it is to execute and in what order. This data is put into memory with a LOAD or RUN command or is simply typed in from the keyboard.

You might think that an Applesoft program is stored in memory in exactly the same format in which it is displayed when it is listed. To save valuable memory space (an Applesoft program and its variables cannot use up more than about 36,000 bytes when ProDOS is being used), and to speed up program execution, however, each line of an Applesoft program is analyzed and compressed before it is actually inserted into the proper area of memory. This process is called “tokenization” because it involves, among other things, substituting one-byte tokens for Applesoft keywords. For example, if you enter the line

```
100 HGR2
```

it is not stored as nine bytes in memory as it would be if you used a standard line editor to create the source file (eight bytes of text plus one byte for the carriage return that follows the line). Rather, it is stored as six bytes: two for the line number, one for the token for the HGR2 keyword, and three for overhead information (these overhead bytes will be described below).

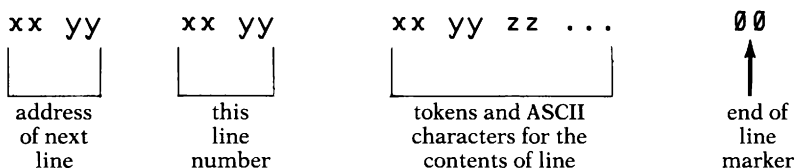
It is the tokenized program that is analyzed by the Applesoft interpreter and not the original source listing. By the way, listing a program is the same as “detokenizing” it because the LIST command essentially converts tokens back into their full keywords.

Let’s take a detailed look at what happens when you add a new line to an Applesoft program while in direct mode (that is, when the program is not running and the “]” prompt symbol is being displayed).

When you type in a line of characters (each line can be up to 239 characters in length) and then press the [return] key to enter it, Applesoft scans the input line and checks to see whether it begins with a valid line number. If it doesn’t, then Applesoft thinks that this is a direct command and attempts to execute it right away; if it does begin with a line number, then Applesoft usually interprets it as a deferred command (that is, one that is to be executed only when the program is executed) and will tokenize it and store it in the proper position in memory. However, if a valid line number is entered by itself, the Applesoft interpreter will delete that line in the program.

The line is placed in memory in such a way that the ascending numeric sequence of the line numbers in the program is maintained. The lowest-numbered line is stored lowest in memory at the location pointed to by the beginning-of-program pointer, TXTTAB(\$67), and the higher-numbered lines are stored sequentially upward in memory.

The bytes that make up a tokenized line are arranged in memory as follows:



The "address of next line" and "this line number" fields are stored as two bytes, with the least-significant byte coming first. The three bytes of overhead that were mentioned above are made up of the two bytes allocated for the address of the next line and the 00 byte that marks the end of the line.

Keyword Tokens

We will now take a closer look at what the tokenized part of the line (the part between the line number and end-of-line marker) looks like. We will begin with a description of the tokens used to replace the Applesoft keywords in a program line. These keywords represent the Applesoft commands, functions, and mathematical and logical operators.

Each Applesoft keyword is assigned by the interpreter to a one-byte quantity called a token. This is done for two main reasons: first, to conserve memory space and, second, to improve the execution speed of the program.

The tokens that Applesoft assigns to each of its keywords are presented in Table 4-2 together with the addresses of the subroutines within Applesoft that are used to deal with the keyword command or function that they represent (where applicable). You will notice that all of these tokens are greater than or equal to \$80. If the tokenized part of a program line contains bytes that are less than \$80, then these bytes are simply the ASCII codes for the characters that were typed in when the line was entered. (See Appendix I for the ASCII codes used to represent characters.) This will include all digits (other than those entered for the line number), all text between quotation marks after PRINT, DATA, and REM statements, and all characters used to represent variable names.

Before you get hopelessly confused, let's look at an example. From Applesoft direct mode, enter NEW, and then enter the following line:

```
100 PI = 4 * ATN (1): PRINT "PI = ";PI: END
```

The bytes used to store this line in memory are as follows: (You can see these bytes for yourself by first entering CALL -151 to enter the system monitor, and then entering 801.81C to display the first few bytes of the program. As we saw earlier, an Applesoft program is usually stored in memory beginning at location \$801.)

Table 4-2. Applesoft keyword tokens. (continued)

Token	Keyword	Address of Subroutine
\$80	END	\$D870
\$81	FOR	\$D766
\$82	NEXT	\$DCF9
\$83	DATA	\$D995
\$84	INPUT	\$DBB2
\$85	DEL	\$F331
\$86	DIM	\$DFD9
\$87	READ	\$DBE2
\$88	GR	\$F390
\$89	TEXT	\$F399
\$8A	PR#	\$F1E5
\$8B	IN#	\$F1DE
\$8C	CALL	\$F1D5
\$8D	PLOT	\$F225
\$8E	HLIN	\$F232
\$8F	VLIN	\$F241
\$90	HGR2	\$F3D8
\$91	HGR	\$F3E2
\$92	HCOLOR =	\$F6E9
\$93	HPLOT	\$F6FE
\$94	DRAW	\$F769
\$95	XDRAW	\$F76F
\$96	HTAB	\$F7E7
\$97	HOME	\$FC58
\$98	ROT =	\$F721
\$99	SCALE =	\$F727
\$9A	SHLOAD	\$03F5
\$9B	TRACE	\$F26D
\$9C	NOTRACE	\$F26F
\$9D	NORMAL	\$F273
\$9E	INVERSE	\$F277
\$9F	FLASH	\$F280
\$A0	COLOR =	\$F24F
\$A1	POP	\$D96B
\$A2	VTAB	\$F256
\$A3	HIMEM:	\$F286
\$A4	LOMEM:	\$F2A6
\$A5	ONERR	\$F2CB
\$A6	RESUME	\$F318
\$A7	RECALL	\$03F5
\$A8	STORE	\$03F5
\$A9	SPEED =	\$F262
\$AA	LET	\$DA46
\$AB	GOTO	\$D93E
\$AC	RUN	\$D912

(continued)

Table 4-2. Applesoft keyword tokens (continued).

<i>Token</i>	<i>Keyword</i>	<i>Address of Subroutine</i>
\$AD	IF	\$D9C9
\$AE	RESTORE	\$D849
\$AF	&	\$03F5
\$B0	GOSUB	\$D921
\$B1	RETURN	\$D96B
\$B2	REM	\$D9DC
\$B3	STOP	\$D86E
\$B4	ON	\$D9EC
\$B5	WAIT	\$E784
\$B6	LOAD	\$03F5
\$B7	SAVE	\$03F5
\$B8	DEF	\$E313
\$B9	POKE	\$E77B
\$BA	PRINT	\$DAD5
\$BB	CONT	\$D896
\$BC	LIST	\$D6A5
\$BD	CLEAR	\$D66A
\$BE	GET	\$DBA0
\$BF	NEW	\$D649
\$C0	TAB(
\$C1	TO	
\$C2	FN	
\$C3	SPC(
\$C4	THEN	
\$C5	AT	
\$C6	NOT	
\$C7	STEP	
\$C8	+	
\$C9	-	
\$CA	*	
\$CB	/	
\$CC	^	
\$CD	AND	
\$CE	OR	
\$CF	>	
\$D0	=	
\$D1	<	
\$D2	SGN	\$EB90
\$D3	INT	\$EC23
\$D4	ABS	\$EBAF
\$D5	USR	\$000A
\$D6	FRE	\$E2DE
\$D7	SCRN(\$D412
\$D8	PDL	\$DFCD
\$D9	POS	\$E2FF

(continued)

Table 4-2. Applesoft keyword tokens (continued).

<i>Token</i>	<i>Keyword</i>	<i>Address of Subroutine</i>
\$DA	SQR	\$EE8D
\$DB	RND	\$EFAE
\$DC	LOG	\$E941
\$DD	EXP	\$EF09
\$DE	COS	\$EFEA
\$DF	SIN	\$EFF1
\$E0	TAN	\$F03A
\$E1	ATN	\$F09E
\$E2	PEEK	\$E764
\$E3	LEN	\$E6D6
\$E4	STR\$	\$E3C5
\$E5	VAL	\$E707
\$E6	ASC	\$E6E5
\$E7	CHR\$	\$E646
\$E8	LEFT\$	\$E65A
\$E9	RIGHT\$	\$E686
\$EA	MID\$	\$E691

```

1D 08 64 00 50 49 D0 34 CA E1 28 31 29 3A
address line P I token 4 token token
of next number ( 1 )
line
BA 22 50 49 20 3D 20 22 3B 50 49 3A 80 00
token P I = P I token end
for PRINT for line
END marker

```

Notice that the five keywords in this line, =, *, ATN, PRINT, and END, have been replaced by their tokens, \$D0, \$CA, \$E1, \$BA, and \$80, respectively. Also notice that each character that is not part of a keyword is not tokenized and is represented by its ASCII code.

Storage of Applesoft Variables

Now that we have seen how an Applesoft program is stored in memory, let's take a more detailed look at how and where the program's variables are stored during program execution. Not only is the knowledge of the data structures used to store variables fundamentally interesting, it will undoubtedly be invaluable to those who wish to manipulate Applesoft variables from within 65C02 assembly-language subroutines that are called from Applesoft.

Applesoft supports four fundamental variable types. There are three numeric types (integer, real, and function) and one alphanumeric type (string). Integer numbers are made up of all positive and negative whole numbers and zero, that is, all numbers that have no fractional parts (Applesoft handles all those integers between -32767 and 32767). Real numbers, also called floating-point numbers, are made up of all numbers, including those that do have fractional parts. Strings are simply sequences of characters; the characters are encoded using the ASCII scheme (see Chapter 6). Functions are special variables that are defined by the Applesoft DEF FN command and that are evaluated using a user-specified mathematical expression. For example, if a function is defined as follows:

```
DEF FN MD(X)=X-256*INT(X/256)
```

then whenever the value of MD(aexpr) is requested (where “aexpr” represents an arithmetic expression) it is evaluated by substituting the value of “aexpr” wherever “X” appears in the “X-256*INT(X/256)” formula and then calculating the result.

The first character of an Applesoft variable name must begin with a letter from A . . . Z (you can enter it in upper- or lowercase on the //c—Applesoft will automatically convert it to uppercase); subsequent characters can be either letters or a digit from 0 . . . 9. The variable name can be up to 239 characters in length, but only the first two characters are significant (the rest are simply ignored). This means that Applesoft considers the variables LESS and LESSEN, for example, to be equivalent.

A variable name cannot be used that contains the names for any of the keywords shown in Table 4-2. For example, the variable name “LETTER” is illegal because it contains the LET keyword.

If an integer or string variable is being defined, a special variable identifier symbol must be added to its name so that Applesoft can properly interpret it and store its value. The variable identifier symbol for integer variables is “%” and for string variables it is “\$”. No special identifier symbol is needed to identify real or function variables. Table 4-3 sets out the variable identifier symbols used by Applesoft.

When a variable is defined in a program, Applesoft stores its name and value at the end of one of two memory spaces located after the end of the program. One space is reserved for simple variables and functions and is pointed to by VARTAB (\$69). The other space is reserved for array variables and is pointed to by ARYTAB (\$6B). In the following sections, we will take a look at how variables are represented in these two variable spaces.

Storage of Simple Variables

Whenever Applesoft has to make use of a certain variable, it has to locate it within its variable space. It does this by searching the variable space

Table 4-3. Applesoft variable identifier symbols.

Variable Identifier Symbol	Variable Type	Example
[none]	real	AB
%	integer	AB%
[none]	function	FN AB()
\$	string	AB\$

beginning with the first entry and continuing until it finds a match. Thus, the farther into the space a variable is located, the longer it will take Applesoft to find it. Since Applesoft stores variables in its variable space in the order in which they are encountered when the program is executed, you can improve program execution speed by ensuring that more frequently used variables are defined before less frequently used ones. This is most easily done by defining all the frequently-used variables in the desired order as soon as the program starts executing. For example, if your program uses four variables, say I, J, K, and L\$, but you would like K to be accessed as quickly as possible, then you should execute a line such as

```
10 K=0:I=0:J=0:L$=""
```

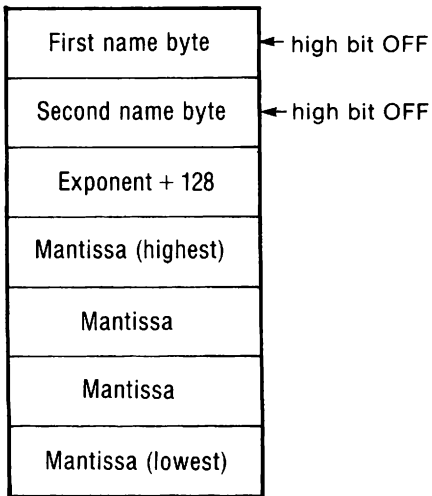
before any other line that defines or uses any variables.

Each entry in the simple variable space is exactly seven bytes long and consists of two parts: the name header, which is used to store the variable's name and type, and the data field, which contains the encoded value of the variable or a pointer to its location. The storage format used for each type of variable is summarized in Figure 4-2.

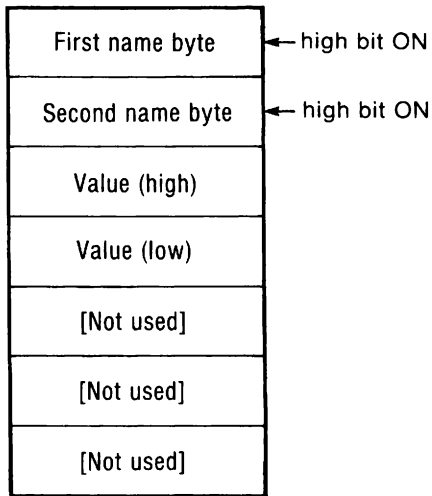
The Name Header

The name header contains all the information related to the variable's type and name so that it can be quickly located and accessed whenever it is referred to during execution of the Applesoft program. The name header for a simple variable is always exactly two bytes long. Stored in these two bytes are the 7-bit ASCII codes for the first two characters of the variable's name; if there is only one character used in the name, then the second character is assumed to be the ASCII null character, \$00. The high-order bits of each of the two bytes are used to indicate the type of simple variable being referred to. For example, for a string variable, these bits will be OFF (0) and ON (1), respectively. For real and integer variables, they will be OFF-OFF and ON-ON, respectively. Lastly, the bits will be ON-OFF if the name refers to a function defined by the DEF FN command.

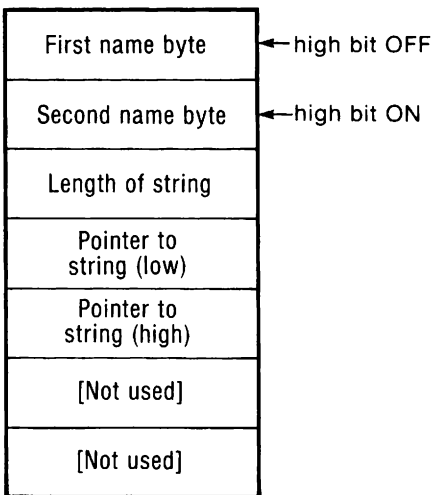
(a) Real variables.



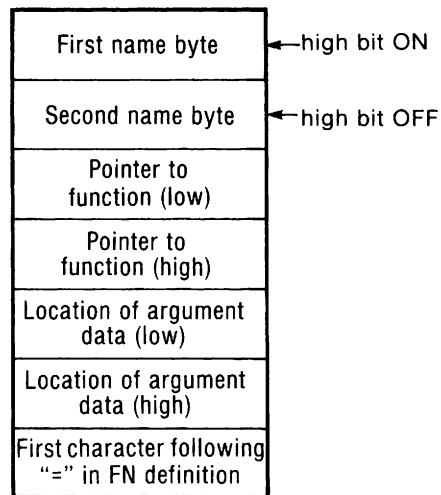
(b) Integer variables.



(c) String variables.



(d) Function variables.

**Figure 4-2. Storage formats for Applesoft simple variables.**

The Data Field

The encoded data that relates to the value of the simple variable are stored in five bytes just after the end of the two name header bytes. Despite the fact that five bytes are always reserved for data storage, however, only real variables and functions make use of them all. The number of bytes required for

the data for each type of variable is as shown in Table 4-4, as are the restrictions on the values for each type of Applesoft variable.

Table 4-4. Storage requirements and limitations for Applesoft variables.

<i>Variable Type</i>	<i>Number of Data Bytes Required</i>	<i>Restrictions on Variable Value</i>
Integer	2	$-32767 \dots +32767$
Real	5	$2.9\text{E}-39 \dots 1.7\text{E}+38$ (pos. or neg.)
String	3	Length of string is $0 \dots 255$
Functions	5	One argument only

Let's take a look at the storage formats used for each type of variable.

Integer. The data for integer variables is stored in a signed "two's complement" format and occupies two bytes (most-significant byte followed by least-significant byte). See the section below entitled "Representation of Integer Numbers" for a detailed description of the two's complement storage format. The high bit of the most-significant byte can be read to determine the sign of the number. If this bit is 1, then the number is negative; if it is 0, then the number is positive. The last three bytes of the data field are not used.

Real. The data for real numbers is stored in all five bytes. The first byte is related to the exponent of the number and the next four bytes represent its signed mantissa, most-significant byte first. The sign bit is the high bit of the second byte of the five. See the section below entitled "Representation of Real Numbers" for a detailed description of the method Applesoft uses to store real numbers.

String. The data for string variables is really made up of two parts. The first part is stored in the variable table itself and is a three-byte "descriptor" that represents the length of the string (first byte) followed by a two-byte pointer (low-order byte first) to a sequence of ASCII-encoded characters that defines the string itself. The second part is, in fact, made up of those characters that define the contents of the string.

The contents of strings are normally stored in the high end of memory in a string space beginning at a location pointed to by MEMSIZ (\$73) and ending lower in memory just before the location pointed to by FRETOP (\$6F). Whenever a new string is entered from the keyboard or a diskette file, or an old one is manipulated using any of Applesoft's string-handling commands, it is placed in memory just before the address to which FRETOP points in such a way that the first character in the string is located lowest in memory and the last

character is located at the location pointed to by FRETOP. After this is done, FRETOP is adjusted downward so that it points to the byte immediately before the beginning of the string just stored.

When a string variable is redefined using Applesoft's string-handling commands, its new definition is placed in the string space in the upper part of memory as if it were a newly defined variable; however, its former characters are not immediately removed from the string space even though it is no longer used. This means that if strings are continuously being redefined, a lot of unused information will accumulate in the string space and eventually the address stored in FRETOP will come very close to the address stored in the end-of-variable pointer, STREND (\$6D).

When this happens, a procedure is initiated that maximizes the available free space by removing the unused string characters, packing the currently active string characters up to the high end of memory, and resetting FRETOP. This procedure is called "garbage collection" or, more euphemistically, "house-cleaning". Applesoft's own garbage collection routine can last anywhere from a few seconds to a few minutes, depending on the number of string variables that have been defined in the program. However, when an Applesoft program is running in a ProDOS environment, a garbage collection routine within the BASIC.SYSTEM program that interfaces Applesoft to ProDOS (see Chapter 5) is used instead and it handles the collection virtually instantaneously.

Note, however, that if a string is explicitly defined within the program itself, for example, in a program line that looks like this:

```
100 A$="THIS IS A TEST"
```

then the string pointer in the variable's data field will point to the definition inside the program itself and not to a location within the usual string space. Such a string will be moved into the string space only if it is operated on by an Applesoft string-handling command.

Functions. The data for functions is stored in five bytes. The first two bytes act as a pointer to the body of the function's definition within the program (that is, the part after the "=" sign in the DEF FN definition). The next two bytes contain the address of the data field for the variable representing the function's argument. The last byte contains the first byte in the function definition.

End of Simple Variables

ARYTAB (\$6B) points to the Applesoft array variable space located immediately after the end of the simple variable space. Whenever a new simple variable is defined, the whole of the array variable space is moved up in memory by seven bytes to make room for the new simple variable definition and the end-of-variables pointer, STREND (\$6D), is adjusted accordingly.

The name header and data bytes for the variable are then stored beginning at ARYTAB. ARYTAB is then increased by seven so that it equals the new starting position of the array space.

The simple and array variable spaces are also moved upward and downward in memory as program lines are added to or deleted from the program.

Storage of Array Variables

Each entry in the array variable space is made up of a name header, special dimensioning bytes that indicate the size of the array and how it is indexed, and a data field. The storage format used for each type of array variable is summarized in Figure 4-3. Note that arrays are permitted for each Applesoft variable type except functions.

The Name Header

Just as for simple variables, entries for array variables begin with a name header. The name headers for array variables are identical to those for the corresponding simple variables discussed in the previous section (for example, the header for an array dimensioned as AB(5,6) is the same as for AB).

Dimensioning Bytes

When array variables are stored, a series of bytes that describe the number of dimensions of the array and their sizes are placed in memory just after the header.

First, two bytes are used to store a number that is equal to the number of bytes that the array occupies in the array variable space. This number is simply the offset from the name header of this array to the next array and is stored here so that the address of the next array variable in the array space can be quickly and easily calculated when Applesoft is searching for an array. The number is stored with the low-order byte first.

The next byte is equal to the number of array indexes (or “dimensions”) and can be from 1 to 255. For example, an array dimensioned as AB(3,5,2) would have a value of 3 stored in this byte.

Pairs of bytes follow this last byte that indicate the size of the indexes of the array, with the number of elements in the last index being stored in the first pair and the number of elements in the first index being stored in the last pair. The high-order byte is stored first in each pair. The numbers stored here will be one higher than the number used when the array was first dimensioned (using the DIM statement) since it starts counting the elements from 1 rather than 0.

Let's look at an example. The name header bytes and dimensioning bytes for an array dimensioned as AB(3,5,2) would be as follows:

41 42	73 01	03	00 03	00 06	00 04
<div style="border: 1px solid black; width: 40px; height: 15px; margin: 0 auto;"></div>	<div style="border: 1px solid black; width: 40px; height: 15px; margin: 0 auto;"></div>	↑	<div style="border: 1px solid black; width: 40px; height: 15px; margin: 0 auto;"></div>	<div style="border: 1px solid black; width: 40px; height: 15px; margin: 0 auto;"></div>	<div style="border: 1px solid black; width: 40px; height: 15px; margin: 0 auto;"></div>
name (AB)	offset to next array	# of indexes	size of 3rd index	size of 2nd index	size of 1st index

Header used by all three array variable types:

First name byte
Second name byte
Offset to next array variable (low byte first)
Number of dimensions
Size of last dimension (high byte first)
//
Size of first dimension (high byte first)

(a) Real variables.

Exponent + 128
Mantissa (high)
Mantissa
Mantissa
Mantissa (low)
//
Exponent + 128
Mantissa (high)
Mantissa
Mantissa
Mantissa (low)

} first element

} last element

(b) Integer variables.

Value (high)
Value (low)
//
Value (high)
Value (low)

} first element

} last element

(c) String variables.

Length of string
Pointer to string (low)
Pointer to string (high)
//
Length of string
Pointer to string (low)
Pointer to string (high)

} first element

} last element

NOTE: Array elements are stored in such a way that the right-most dimensioning index increases slowest (see text).

Figure 4-3. Storage formats for Applesoft array variables.

The Data Field

After the dimensioning bytes come the actual data bytes for each array element. They are stored in exactly the same formats used by the corresponding simple variables except that, in the case of integer and string arrays, the data bytes are packed. This means that the unused bytes that are stored in the simple variable data space for these two types of variables are not stored.

The array elements are stored in memory in such a way that the rightmost dimensioning index ascends most slowly. Thus, if an array is dimensioned as $A(1,1)$, then $A(0,0)$ is stored first, followed by $A(1,0)$, $A(0,1)$, and then $A(1,1)$.

End of Array Variables

STREND (\$6D) points to one byte past the end of the array variable space. It also points to the beginning of Applesoft free space. When a new array variable is defined, its header and data are stored beginning at this location and then the value STREND is increased by the size of the entry for the array.

Representation of Integer Numbers

Applesoft stores the data for its integer variables in a special two-byte format called "two's complement." As we will see, the advantage of using this format is that it allows both negative and positive numbers to be represented in a way that greatly simplifies the execution of the two basic signed arithmetic operations, addition and subtraction.

The most-significant byte of the pair of data bytes reserved for an integer is stored first (note that this is just the opposite of how two-byte quantities are usually stored). The high-order bit of this byte is used to indicate the sign of the number. If it is 1, then the number is negative; if it is 0, then it is positive. The remaining 7 bits of this byte, and the 8 bits of the least-significant byte, are used to represent the magnitude of the integer. For a positive integer, the 15-bit magnitude is simply represented by the standard unsigned binary pattern for the integer. For example,

00000001 00000011

is used to represent +259 (\$0103).

The 15 bits used to represent a negative integer are determined somewhat differently. To determine what they are, you must first take the binary pattern for the absolute value of the integer (that is, its positive counterpart), complement it by changing all its 1 bits to 0 and vice versa, and then add one to the result. The most-significant bit will then be 1, indicating that the number

is negative. For example, the representation for the integer -11 would be calculated as follows:

$$\begin{array}{r}
 00000000\ 00001011\ (+11) \\
 11111111\ 11110100\ (\text{complement}) \\
 + \quad \underline{\hspace{1cm}1\hspace{1cm}}\ (\text{add } 1) \\
 11111111\ 11110101\ (-11 \text{ in two's complement})
 \end{array}$$

Using the two-byte two's complement format, it is possible to define integers that range from -32768 (100000000 00000000) to $+32767$ (01111111 11111111). Note, however, that even though the number -32768 can be represented in the two-byte two's complement format, Applesoft does not allow its integer variables to take on this value. The lowest value that is allowed is -32767 .

Applesoft stores its integers in this apparently strange format to simplify the way in which binary arithmetic can be performed. By using the two's complement format, positive and negative numbers can be easily added and subtracted without having to perform the complicated adjustments needed to account for the different signs of the numbers if any other representation is used. (Another representation may be the conventional "sign plus magnitude" (S + M), where a positive integer and its negative counterpart are identical except for the value of the sign bit.) When using the two's complement representation, it is only necessary to add the 16-bit representations of the two integers (be they positive or negative) as if they were just two standard unsigned binary numbers. The result, and its sign, will then automatically be correct if the result is viewed as another two's complement integer (which it is).

Let's take a look at an example to see what we mean by this. Consider the problem of adding the integer $+8$ to the integer -5 . If these numbers were stored in their normal binary representations with the sign bit being the most-significant bit, then the calculation to be performed would be

$$\begin{array}{rcl}
 & 00000000 & 00001000 & (+8) \\
 + & \underline{10000000} & 00000101 & (-5 \text{ in S+M binary}) \\
 & 10000000 & 00001101 & (-13 \text{ in S+M binary})
 \end{array}$$

This result is, of course, wrong. Thus, if this representation is used, special programs must be written to avoid these erroneous results. On the other hand, if the integers are represented in the two's complement format, then the calculation becomes

$$\begin{array}{r}
 00000000 \ 00001000 \quad (+8) \\
 + \quad 11111111 \ 11111011 \quad (-5 \text{ in two's complement}) \\
 \hline
 00000000 \ 00000011 \quad (+3)
 \end{array}$$

This result is, of course, correct. If you experiment with other integers, you will see that the signed result is always correct (unless the result is out of the allowable range).

Representation of Real Numbers

As we have seen, Applesoft real numbers are stored in the simple variable space and array variable space in a binary floating-point format. This special format will be described in detail now.

Knowledge of this format will be of use mainly to those who write 65C02 assembly-language programs that access Applesoft numeric variables. However, even if you never intend to write such a program, the following information should prove to be interesting.

Number Theory

Even though numbers are commonly entered into a computer in a “decimal” or “base 10” format, they are generally stored internally in some sort of compressed binary format to reduce data storage space and to make it easy for programs to manipulate them.

Decimal integer numbers can be stored in a binary form without loss of accuracy due to rounding or truncation (provided that the integers are within the numeric range supported by the computer) because they do not contain fractional parts. On the other hand, floating-point numbers (that is, real numbers), which do have fractional parts, can only be “approximated” by a binary representation unless the decimal number is exactly equal to a sum of powers of two. Because approximations have to be made in most cases, you will sometimes find that if you multiply a number by its reciprocal in Applesoft that the number calculated is not equal to one!

Floating-point real numbers are often expressed in “scientific notation” that looks like this:

$$134.56 \times 10^6$$

The first part of this representation is called the mantissa and the second part is called the exponent (the exponent is actually the number to which the number base being used has been raised). An understanding of scientific notation is important because it is a binary mantissa and exponent that are stored by Applesoft when real numbers are saved in its variable spaces.

Binary Floating-Point Format

Real numbers are stored in the variable spaces of Applesoft in a “binary floating-point” format. As indicated in Figure 4-4, this is a five-byte format in which one byte is reserved for exponent information and four bytes for mantissa information. The mantissa contains the binary representation of the fractional part of the number.

The lowest-addressed byte in the fivesome is the exponent byte. The value

stored here is actually not the exponent itself but rather the value of the exponent plus 128. Because this method is used to store the exponent, the exponent is said to be “biased” by 128.

Before a number is stored in the binary floating-point format, it is “normalized.” Normalization is the process whereby the binary point of the binary number (as opposed to a decimal point for a decimal number) is adjusted so that there is a “1” to its immediate right and no “1”’s to the left of it. Thus, after normalization, the mantissa of the number will be between 0.1 and 0.1111111 . . . (in binary). For each movement of the binary point to the left, the exponent is increased by one; for each movement to the right, the exponent is reduced by one. For example, consider the binary number “1101.11”. To normalize this number, the binary point must be moved four places to the left; thus, the initial exponent (0) must be increased by four.

In the binary floating-point representation, the high-order bit of the second byte represents the sign of the number. If this bit is 1, then the number is negative; if it is 0, then the number is positive.

The remaining 7 bits of the second byte and the remaining three bytes are used to represent the mantissa of the number, most-significant byte first. Within a particular byte, the 7th bit is the most significant and the 0th bit the least significant. As has been explained, the mantissa has been normalized so that there is a “1” to the immediate right of the decimal point; this “1” is implicit and is not stored. Thus, a floating-point number has 32-bit precision (about nine decimal digits) even though only 31 bits are actually used to hold the mantissa.

Any number whose exponent byte is equal to zero is considered to be zero by the Applesoft interpreter even though its mantissa bytes may be nonzero.

The decimal range of numbers that is allowed using the five-byte binary floating-point format is as follows:

+/- 2.9387355E-39 to +/- 1.70141183E+38

To calculate a decimal number from its binary format, multiply the value of each mantissa bit by its corresponding binary weight, add the implied 0.5 (which is the decimal equivalent of binary 0.1), and then multiply the total by 2 raised to the value of the exponent byte minus 128. The binary weight of a particular mantissa bit is given by $(1/2)^{(32-BN)}$, where BN is the bit number. The bit numbers range from the most-significant bit 30 (bit 6 of byte 2) to the least-significant bit 0 (bit 0 of byte 5).

For example, consider the decimal number ‘8.67’. It is stored by Applesoft as the following five bytes:

<u>BYTE1</u>	<u>BYTE2</u>	<u>BYTE3</u>	<u>BYTE4</u>	<u>BYTE5</u>
\$84	\$0A	\$B8	\$51	\$EB

and the corresponding binary number is

$$+ . 1 \text{ } 0001010 \text{ } 10111000 \text{ } 01010001 \text{ } 11101011 \text{ } * 2^{84-80}$$

$$\uparrow \text{ byte2 } \quad \text{byte3} \quad \text{byte4} \quad \text{byte5} \quad \text{byte1}$$
 implicit

To convert this binary number to its corresponding decimal number, you must add the implicit 0.5 to the sum of each binary digit multiplied by its binary weight. The resultant calculation is as follows:

$$\begin{aligned}
 &0.5 + (1/2)^5 + (1/2)^7 + (1/2)^9 + (1/2)^{11} + (1/2)^{12} \\
 &\quad + (1/2)^{13} + (1/2)^{18} + (1/2)^{20} + (1/2)^{24} + (1/2)^{25} \\
 &\quad + (1/2)^{26} + (1/2)^{27} + (1/2)^{29} + (1/2)^{31} \\
 &\quad + (1/2)^{32}
 \end{aligned}$$

If you calculate this quantity, you will get 0.541875. It then must be multiplied by the exponential part (which is 2^4 or 16) in order to yield the final result: 8.67.

Note that the high bit of BYTE2 in the above example is zero indicating that the number is positive.

If you wish to look at the bytes that Applesoft uses to store other numbers, use the program found in Table 4-5. When you RUN this program, you will be asked to enter a number to be analyzed (X). The program locates the data bytes used to store this number by recognizing the fact that since X is the first simple variable defined in the program, its five data bytes must be stored two bytes from the beginning of the simple variable space (remember that the first two bytes are reserved for the name header). The address of the beginning of this space is simply $\text{PEEK}(105) + 256 * \text{PEEK}(106)$ since the pointer to the beginning of the simple variable space is located at \$69 and \$6A.

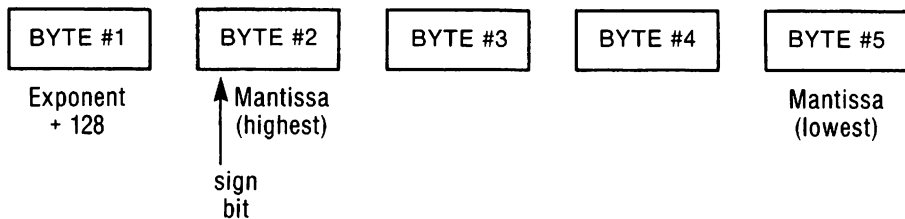


Figure 4-4. Applesoft binary floating-point format.

How an Applesoft Program Runs

Right after you enter the RUN command to begin execution of an Applesoft program, at least two important things happen. First, all the pointers to the variable spaces are initialized, effectively destroying any variables that may have been active when the program last stopped running. Then, just before the program starts to be executed, a special pointer, called TXTPTR (\$B8/

Table 4-5. REAL.NUMBERS—a program to display the bytes that are used to represent an Applesoft real variable.

```

0  REM "REAL.NUMBERS"
100 TEXT : HOME : PRINT "DECIMA
    L ---> BINARY FLOATING-POINT
    "
110 VTAB 5
120 INPUT "ENTER NUMBER TO BE C
    ONVERTED: ";X
130 DIM HX$(15): FOR I = 0 TO 1
    5: READ HX$(I): NEXT
140 XD = PEEK (105) + 256 * PEEK
    (106) + 2: REM LOCATION OF
    DATA FOR X
150 PRINT : PRINT "THE FLOATING
    -POINT REPRESENTATION IS:": PRINT

160 FOR I = XD TO XD + 4
170 D = PEEK (I):D1 = D
180 PRINT "BYTE #";I - XD + 1;"
    : ";
190 FOR J = 7 TO 0 STEP - 1
200 T = INT (D / (2 ^ J)): PRINT
    T;
210 D = D - T * (2 ^ J)
220 NEXT J: PRINT " ($";HX$( INT
    (D1 / 16));HX$(D1 - 16 * INT
    (D1 / 16));" ) ";
230 READ DS$: PRINT DS$
240 NEXT I: PRINT
250 PRINT "BIT 7 OF BYTE #2 IS
    THE SIGN BIT": PRINT "(0 -->
    POSITIVE, 1 --> NEGATIVE)"
260 DATA 0,1,2,3,4,5,6,7,8,9,A,
    B,C,D,E,F
270 DATA EXPONENT + 128,MANTISS
    A HIGH,...,MANTISSA LOW

```

\$B9), is initialized so that it contains the address of the beginning of the program. This address is normally \$801.

TXTPTR is an important pointer as far as the interpreter is concerned because it always contains the address of the location within the program that the interpreter is acting on. Whenever the interpreter wants to examine the next byte of the tokenized program, it simply increments this pointer and then reads the new byte to which it points.

The CHARGET Subroutine

Since TXTPTR must be incremented by many different subroutines in the interpreter, one special subroutine is used to take care of it. This subroutine is called CHARGET (for CHARacter GET) and is located in page zero from location \$B1 to location \$C8. A source listing of CHARGET appears in Table 4-6. Another subroutine, called CHARGOT, is contained within CHARGET; this subroutine reads the current byte being pointed to without incrementing TXTPTR. An image of the CHARGET subroutine is loaded into its page zero locations from the Applesoft ROM area by the Applesoft interpreter when Applesoft is first initialized. It must be placed in a RAM area because, as we will see, it contains self-modifying code.

TXTPTR is actually located within this subroutine at location \$B8/\$B9 and it forms the operand of an LDA instruction that retrieves the value of the byte pointed to by TXTPTR.

When CHARGET is called, TXTPTR is incremented, the 65C02 accumulator is loaded with the byte located at the new address it points to, certain processor flags are set, and then the routine ends. Exactly how the flags are set depends on the value of the byte loaded into the accumulator. If it is an end-of-line marker (0) or end-of-statement byte (\$3A), then the zero flag (Z) is set; otherwise, it is cleared. In addition, if the byte is a digit (that is, its ASCII code is between \$30 and \$39), then the carry flag (C) will be clear; otherwise, it will be set. The reason for testing for these conditions in the CHARGET subroutine is that many of Applesoft's internal subroutines are constantly checking for end-of-line conditions or for the presence or absence of numbers and this is an efficient way of providing that information. If it wasn't done this way, then wasteful duplication of code would be required because every subroutine that needed the information would have to perform its own separate testing procedures.

Let's get back to our program, which was just starting to run with TXTPTR set to \$801 when we last left it. Since the first four bytes of the program (\$801 . . . \$803) are simply the line number and the address of the next line, they are skipped over by increasing TXTPTR by four so that the next time CHARGET is called the first byte in the token field of the program line will be read.

The next step, of course, is to call CHARGET and get that first byte and analyze it. This is where Applesoft really starts its interpretation chores. If the byte happens to be an end-of-line marker (0), then TXTPTR is bumped by four positions so that it points to the byte just before the token field of the next line. If it's a colon separator (\$3A), then CHARGET is called again to load the next byte (which will be the first byte in the token field of the next statement on that line).

If the byte is a keyword token (that is, it is greater than or equal to \$80), then, assuming it is not out of context, the appropriate subroutine in the

Table 4-6. CHARGET—the subroutine that is used to parse an Applesoft program.

```

*****
* CHARGET *
*****
1          TXTPTR      EQU    $B8           ;(NOTE: This is CHARGOT+1)
2
3          ORG         $B1
4
5          CHARGOT     INC    TXTPTR       ;Bump the text pointer
6                                     ; by one position
7
8          BNE         CHARGOT             ;Get the byte pointed to
9                                     ; and compare it to ":"
10        INC         TXTPTR+1            ;Branch if ">=":"
11
12        CHARGOT     LDA    $FFFF        ;Is this a blank?
13
14        CMP         # $3A              ;Yes, so get next byte
15        BCS         EXIT               ;No
16        CMP         # $20              ;No
17        BEQ         CHARGOT            ;Yes, so get next byte
18
19        SEC         # $30
20
21        SBC         # $D0
22
23        RTS

```

interpreter that handles that command or function to which it refers will be called. That subroutine will, among other things, evaluate numerical or string expressions and perform syntax checking; it will do this by making extensive use of CHARGET to analyze the bytes “surrounding” the keyword. When the keyword has been dealt with, CHARGET will point to the next byte to be interpreted.

If the byte is not a keyword token or an end-of-line or end-of-statement marker, then, depending on the context, it may be considered to be a variable name, a piece of data, or maybe nothing at all (in which case you will see the dreaded ?SYNTAX ERROR). As long as no syntax errors are detected, TXTPTR will continue incrementing and interpreting new bytes until such time as the token for END or STOP is encountered or until the last line in the program, denoted by a pair of zero bytes where the next line pointer would normally be found, has been executed.

Changing Program Flow

Because Applesoft always relies on the value of TXTPTR to determine what part of the program to execute next, you can easily cause Applesoft to skip certain parts of the program and to continue executing elsewhere merely by adjusting TXTPTR. In fact, this is exactly how the Applesoft GOTO and GOSUB commands work. When the interpreter encounters either of these commands, it performs a number of tasks, the most important of which are to determine the target line number, to find that line number in memory, and then to store the address of the line’s token field in TXTPTR. Then, when Applesoft continues interpreting the program by calling CHARGET, the commands there will begin to be executed.

Finding Line Numbers

We have just seen how TXTPTR is adjusted when either a GOTO or GOSUB command is executed. What we did not explain is how the interpreter determines where the line is located to which control is to be passed by either of these commands.

There are two different methods Applesoft uses, depending on whether the high-order byte of the destination line number is greater than the high-order byte of the current line number. If it is, then the interpreter starts looking for a line with the proper number beginning with the next line in memory. If it is not, then the interpreter begins with the first line of the program. The interpreter can quickly skip over lines whose numbers don’t match by examining the link field address (the first two bytes of the tokenized line) to determine the address of the next line of the program.

What this means is that GOTO and GOSUB commands that transfer control

to line numbers just before the current line will execute more slowly than those that transfer control to lines nearer the start of the program or to lines just after the current line.

It should be obvious, then, that to increase program execution speed, “backward” GOTO and GOSUB statements should transfer control to lines that are as close to the beginning of the program as possible. By placing commonly used subroutines near the beginning of a program in decreasing order of activity, program speed can be noticeably increased.

Linking Applesoft to Assembly-Language Programs

The execution speed of an Applesoft program can be improved dramatically by linking it to assembly-language subroutines. This is because the code generated by the assembly process is directly executable by the microprocessor and does not have to be interpreted first. Such subroutines can be accessed from Applesoft by using one of three Applesoft commands: CALL, USR, and & (ampersand). These three commands are summarized in Table 4-7.

Assembly-language subroutines often need to make use of zero page locations to take advantage of some of the 65C02's more powerful addressing modes. As we have seen, however, several locations in zero page are reserved for use by Applesoft pointers. Others are used by Applesoft, the system monitor, or ProDOS for other purposes. Table 2-5 at the end of Chapter 2 contains a complete list of those zero page locations that are not used and that are available for use by an assembly-language program.

Table 4-7. Applesoft to assembly-language commands.

<i>Command</i>	<i>Description</i>
CALL aexpr	Transfers control to the memory location specified by “aexpr”.
X = USR(aexpr)	Evaluates “aexpr” and places the result in the floating point accumulator (see text) and then transfers control to \$000A. On return, the value of the function is set equal to the value in the FAC.
&	Transfers control to \$3F5.

Note: “aexpr” represents an arithmetic expression.

The CALL command

The CALL command is the one that is usually used to link Applesoft programs with assembly-language subroutines. If such a subroutine begins at a memory location represented by "aexpr," then you would use the command

```
CALL aexpr
```

to invoke the subroutine. The value of "aexpr" that you use must be a literal decimal number (not hexadecimal) or, alternatively, a mathematical expression that evaluates to an integer number.

For example, to execute a subroutine from Applesoft that begins at location \$300 (768 decimal), you would use the command

```
CALL 768
```

When the subroutine finishes executing, you will normally return to Applesoft and the next statement in the Applesoft program will be executed.

You can try using the CALL command without even writing any assembly-language subroutines simply by accessing subroutines that are already contained in the system monitor ROM. For example, to clear the screen you would use the command CALL 64600 since \$FC58 is the address of the screen clear command. As explained in Chapter 3, there are many other subroutines in the monitor, some of which require that data be provided to them first or that registers be set up in certain ways.

If the subroutine that you are calling requires that data be provided to it before it can perform its duties, you would normally precede your CALL with several POKE commands that would place the appropriate information at the locations expected by the subroutine. Similarly, you will usually have to use the PEEK command to examine any numerical results that the program may store in memory.

It is possible, however, using more advanced techniques, to pass the values of named variables to and from your called subroutines. These techniques will be described below in the section entitled "Using Applesoft's Built-in Subroutines."

The & Command

The & (ampersand) command is similar to the CALL command and is used for similar purposes. Whenever the Applesoft interpreter comes across the & command, it immediately causes the system to transfer control to location \$3F5, thus causing the subroutine that is located there to be executed. In the usual case, a 65C02 JMP (jump) instruction is stored at this location that passes control to some other location where the main body of the subroutine begins.

If you want to use the & command to access assembly-language subroutines, you must first set up the jump at location \$3F5 (1013) so that it points to the desired subroutine. This can be done by using the following three POKE commands:

```
POKE 1013,76    REM 76 ($4C) is the 65C02 JMP opcode
POKE 1014,YY     REM YY is the low address of the sub.
POKE 1015,XX     REM XX is the high address of the sub.
```

To calculate the high and low halves of the address of the subroutine, you can use the following formulas:

```
XX = INT(ADDRESS/256)
YY = ADDRESS - 256*XX
```

After you install the subroutine at the proper location, you can then execute the & command to access it.

As with the CALL statement, no built-in provisions have been made for the passing of variables to and from & subroutines. However, the program that is called can be written to do this for itself. See the section below entitled "Using Applesoft's Built-in Subroutines."

The USR Function

The USR function can also be used to link Applesoft to assembly-language subroutines. The syntax of the USR function is as follows:

```
Y = USR(aexpr)
```

where "aexpr" represents a mathematical expression that is called the argument of the function. When the USR function is encountered by the interpreter, the formula is evaluated, the result of the evaluation is placed in an internal floating-point accumulator (FAC) in zero page and a jump to location \$000A is performed. By setting up a 65C02 JMP instruction at \$000A, you can transfer control to the beginning of an assembly-language program that has been loaded anywhere in memory.

After the program has finished executing, control will return to Applesoft and the Y variable in the above equation will be set equal to the current value of the FAC. This is why USR is called the "user-defined function."

Let's take a look at a specific application involving the USR command. In particular, let's calculate the sine of the argument by using Applesoft's internal sine evaluation subroutine located at \$EFF1. As we will see later in this chapter, this subroutine calculates the sine of the number in the FAC and returns the result there. The subroutine required to perform the conversion is simple: JMP \$EFF1. You can install it at location \$300 by entering CALL - 151 to enter the system monitor, and then entering the command

```
300:4C F1 EF
```

To link this subroutine to the USR command, a JMP \$300 instruction must be placed at the USR locations from \$A to \$C. This can be done by entering the following monitor command:

```
A:4C 00 03
```

where 4C is the JMP opcode and 00 03 represents the address of the subroutine (low-order byte first). Note that you could have also entered all this information using Applesoft POKE statements.

To try out the USR routine, enter and RUN the following short program after entering the data at \$300 and \$0A to \$0C as discussed above:

```
100 X = 3
200 PRINT USR (X)
300 PRINT SIN (X)
```

As you will see after the program has executed, USR is indeed calculating the sine of X.

USR is not a popular Applesoft function for two main reasons. First, only a single numeric expression can be passed to the USR subroutine. Second, the structure of the internal floating-point accumulator has never been officially described by Apple. However, as we shall see in the section below entitled “Useful Applesoft Built-in Subroutines,” there are many built-in subroutines in Applesoft that can be used to facilitate manipulation of the FAC.

Applesoft’s Built-In Subroutines

The Applesoft interpreter is made up of many subroutines that are used to perform several different functions: evaluating functions, performing arithmetic operations, locating variables, handling errors, and so on. Many of them make use of the previously described CHARGET subroutine and the TXTPTR (\$B8) pointer to perform their duties. Table 4-8 describes some of the more useful and commonly used Applesoft subroutines. The addresses of these subroutines are called “entry points.”

Many of the Applesoft subroutines make use of special locations in the //c’s zero page. The locations that are referred to in connection with the subroutines in Table 4-8 are shown in Table 4-9.

Many of the subroutines contained in Table 4-8 deal with floating-point real numbers. Applesoft uses two seven-byte areas in zero page, one from \$9D to \$A3 and the other from \$A5 to \$AB, to store binary floating-point numbers whenever mathematical operations are being performed on real numbers or functions are being evaluated. These areas are called the primary floating-point accumulator (FAC) and argument register (ARG), respectively. Note

Table 4-8. Applesoft built-in subroutines.**(a) Locating Variables, Data, and Line Numbers**

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$00B1	(177)	CHARGET	Increments TTXPTR by one position and returns the next byte in the program in the A-register. Certain flags are also set: if A is a colon (":") or a zero, then the zero flag is set; otherwise, it is cleared. If A is an ASCII digit ("0" to "9"), then the carry flag is cleared; otherwise it is set.
\$00B7	(183)	CHARGOT	Returns the current byte in the program pointed to by TTXPTR in the A register. The flags are set in the same way as for CHARGET.
\$DFE3	(57315)	PTRGET	Finds the address of the beginning of the data field within the variable space for any Applesoft variable. On entry, TTXPTR must be pointing to the first character of the variable's name. On exit, the address can be found in VARPNT (\$83/\$84) and in Y (high) and A (low).
\$F7D9	(63449)	GETARYPT	Finds the address of the name header for any array variable. On entry, TTXPTR must be pointing to the first character in the variable's name. On exit, the address can be found in LOWTR (\$9B/\$9C).
\$D61A	(54810)	FNDLIN	Locates the line in the program whose number is in LINNUM (\$50/\$51). On exit, if the line is found, the carry flag is clear and LOWTR (\$9A/\$9B) points to the start of the line. If the line was not found, then the carry flag will be set and LOWTR will point to the next higher line.

(continued)

Table 4-8. Applesoft built-in subroutines (continued).**(b) Evaluating Formulas**

Address		Symbolic Name	Description
Hex	(Dec)		
\$DD67	(56679)	FRMNUM	Evaluates a mathematical formula and stores the result in the FAC. On entry, TXTPTR must be pointing to the first character in the formula. On exit, the result is placed in the FAC unless a syntax error is detected in which case an appropriate error message is displayed.
\$E6F8	(59128)	GETBYT	Evaluates a mathematical formula that will yield a result in the range 0 . . . 255. On entry, TXTPTR must be pointing to the first character in the formula. On exit, the result is stored in the X-register and FACLO (\$A1).
\$DD7B	(56699)	FRMEVL	Evaluates a mathematical or string formula and stores the result in the FAC. On entry, TXTPTR must be pointing to the first character in the formula. On exit, if a string formula is being evaluated, \$A0 (low) and \$A1 (high) points to the 3-byte string descriptor.

(c) Converting Numbers

Address		Symbolic Name	Description
Hex	(Dec)		
\$E2F2	(58098)	GIVAYF	Converts the 2-byte signed integer in A (high) and Y (low) into floating-point format and stores it in the FAC.
\$E6FB	(59131)	CONINT	Converts the number in the FAC to a single byte integer. On entry, the number to be converted must be in the FAC. On exit, the single byte integer is contained in the X-register and FACLO (\$A1) unless the result is not in the range 0 . . . 255 in which case an "ILLEGAL QUANTITY ERROR" message is displayed.

(continued)

Table 4-8. Applesoft built-in subroutines (continued).**(c) Converting Numbers**

Address Hex	Address (Dec)	Symbolic Name	Description
\$E752	(59218)	GETADR	Converts the number in the FAC into an unsigned 2-byte integer (0 ... 65535) in LINNUM (\$50/\$51). If the number is negative, then 65535 is added to its value.
\$ED24	(60708)	LINPRT	Converts the unsigned hexadecimal number in X (low) and A (high) into a decimal number and displays it.
\$ED2E	(60718)	PRNTFAC	Prints the number contained in the FAC (in decimal format). The FAC is destroyed by this process.

(d) Applesoft Real-Number Mathematics

Before executing any of the following subroutines, a number must be loaded into the FAC. All of these subroutines first move the number in memory pointed to by Y (high) and A (low) into the ARG and perform the mathematical operation. The result is placed in the FAC.

Address Hex	Address (Dec)	Symbolic Name	Description
\$E7A7	(59303)	FSUB	Subtract the FAC from the ARG.
\$E7BE	(59326)	FADD	Add the FAC to the ARG.
\$E97F	(59775)	FMULT	Multiply the ARG by the FAC.
\$EA66	(60006)	FDIV	Divide the ARG by the FAC.

(e) Applesoft String Handling

Address Hex	Address (Dec)	Symbolic Name	Description
\$E452	(58450)	GETSPACE	Reduces the start-of-strings pointer, FRETOP (\$6F), by the number specified in the A-register (the string length) and sets up FRESPEC (\$71) so that it equals FRETOP. After this has been done, A remains unaffected and Y (high) and X (low) point to the beginning of the space. The string can then be moved into place in upper memory by using MOVESTR.

(continued)

Table 4-8. Applesoft built-in subroutines (continued).**(e) Applesoft String Handling**

Address Hex	Address (Dec)	Symbolic Name	Description
\$E484	(58500)	GARBAGE	Clears out old string definitions that are no longer being used and adjusts FRETOP (\$6F) accordingly. (Each time that a string is redefined, its old definition is kept in memory but is not used.) This process is called "garbage collection" and is performed automatically whenever the start-of-strings address, FRETOP, comes close to the end-of-variables address, STREND (\$6D). Note that under ProDOS this routine is never called since BASIC.SYSTEM does its own faster garbage collection first.
\$E5E2	(58850)	MOVESTR	Copies the string that is pointed to by Y (high) and X (low) and that has a length of A to the location pointed to by FRESPC (\$71).
\$ED34	(60724)	FOUT	Converts the FAC into an ASCII character string that represents the number in decimal form (like Applesoft's STR\$ function). The string is followed by a \$00 byte and is pointed to by Y (high) and A (low) so that STROUT can be used to print the string.
\$DB3A	(56122)	STROUT	Prints the string pointed to by Y (high) and A (low). The string must be followed immediately by a \$00 or a \$22 byte. All of these conditions are set up by FOUT.
\$DB3D	(56125)	STRPRT	Prints the string whose 3-byte descriptor (a length byte followed by a two-byte pointer) is pointed to by \$A0/\$A1. FRMEVL sets up such a pointer when calculating string formulas.

(continued)

Table 4-8. Applesoft built-in subroutines (continued).**(f) Applesoft Real-Number Functions**

In executing the following subroutines, Applesoft expects the argument to be in the FAC. After the result has been calculated, it will be placed in the FAC.

Address Hex	Address (Dec)	Symbolic Name	Description
\$E941	(59713)	LOG	Calculate the natural logarithm
\$EBAF	(60335)	ABS	Calculate the absolute value
\$EE8D	(61069)	SQR	Calculate the square root
\$EF09	(61193)	EXP	Calculate "e to the power of"
\$EFEA	(61418)	COS	Calculate the cosine (in radians)
\$EFF1	(61425)	SIN	Calculate the sine (in radians)
\$F03A	(61498)	TAN	Calculate the tangent (in radians)
\$F09E	(61598)	ATN	Calculate the arctangent (in radians)

(g) Miscellaneous Subroutines

Address Hex	Address (Dec)	Symbolic Name	Description
\$DA0C	(55820)	LINGET	Loads a line number into LINNUM (\$50/\$51). On entry, TXTPTR must point to the first digit of the line number.
\$D412	(54290)	ERROR	Handles any Applesoft error conditions that may occur during the running of a program. The subroutine first checks ERRFLAG (\$D8) to see if an ONERR GOTO statement is in effect; if ERRFLAG >= \$80, then error handling has been enabled and control passes to the appropriate line number. If ERRFLAG < \$80, then an error message is printed (the error-number code is in X) and the program stops.
\$DEBE	(57022)	CHKCOM	Checks that TXTPTR (\$B8) is pointing to a comma and, if it is, bumps TXTPTR by one. If TXTPTR is not pointing to a comma, then a syntax error will be generated.

(continued)

Table 4-8. Applesoft built-in subroutines (continued).**(g) Miscellaneous Subroutines**

Address Hex	(Dec)	Symbolic Name	Description
\$E000	(57344)	COLD	Performs an Applesoft cold start (the program in memory is destroyed).
\$E003	(57347)	WARM	Performs an Applesoft warm start (the program in memory remains intact).

Table 4-9. Some important zero page locations used by Applesoft's built-in subroutines.

Address Hex	(Dec)	Symbolic Name		Description
\$50	(80)	LINNUM	(low)	These are the locations, used by GETADR, that contain the result of the conversion of the FAC to a 2-byte integer.
\$51	(81)		(high)	
\$71	(113)	FRESPC	(low)	This is a temporary pointer, used by GETSPACE and MOVESTR, that contains the address of the location to which a string is to be moved.
\$72	(114)		(high)	
\$83	(131)	VARPNT	(low)	This is a temporary pointer, used by PTRGET, that contains the location of the data bytes for the last variable that was found using PTRGET.
\$84	(132)		(high)	
\$9B	(155)	LOWTR	(low)	A pointer used by FNDLIN and GETARYPT.
\$9C	(156)		(high)	
\$A1	(161)	FACLO		This is a byte in the FAC that contains the result of CONINT and GETBYT.
\$B7	(183)	TXTPTR	(low)	This is a pointer to the position within the program that is currently being acted on by the interpreter. It is part of the CHARGET subroutine.
\$B8	(184)		(high)	
\$D8	(216)	ERRFLAG		This is the ONERR GOTO flag. If it's >= \$80, then ONERR is active.

that despite the use of the words “accumulator” and “register,” these are not 65C02 registers, but merely special data storage areas. Although the format Applesoft uses to store numbers in both FAC and ARG is not quite the same as the five-byte format used to store real numbers in the Applesoft simple and array variable spaces, it will not be described here since knowledge of it is not necessary to make use of Applesoft’s built-in floating-point mathematical subroutines.

The FAC is used by Applesoft to hold the argument for those calculations that require only one argument (for example, the calculation of a sine). If two arguments are required, however, the first argument is kept in the ARG and the second is kept in the FAC. In either case, the answer is returned in the FAC.

Remember the Applesoft USR command? The argument that is evaluated when this command is executed is stored in the FAC, as is the returned result.

Using Applesoft’s Built-In Subroutines

Applesoft’s built-in subroutines can be used in conjunction with your own assembly-language programs to greatly simplify those programs and to allow you to dispense with having to rewrite programs that have already been written. In most cases, it is not even necessary to understand exactly how the Applesoft subroutine operates as long as you understand what the entry conditions are and what effect the subroutine has on the system.

There are literally hundreds of useful subroutines within the Applesoft interpreter that can be accessed, but only a few of them have been listed in Table 4-8. Three of the more important classes of subroutines will be discussed in detail here: those used to locate variables, those used to evaluate formulas, and those used to convert numbers between different formats.

Locating Variables

We have already seen how Applesoft keeps track of its variables and how it stores them. Using that information, it would be a fairly simple chore to write a program to locate and retrieve any particular one. All you would have to do would be to check the name bytes of each variable in the variable table that begins at the start of simple variable space until a match was found. Applesoft already contains a program to do this, however, so why bother? This program is called PTRGET and begins at \$DFE3.

To find the location of a variable, all you must do is adjust TXTPTR (\$B8/\$B9) so that it points to the first character in the variable name, and then execute a JSR PTRGET instruction. When the subroutine ends, VARPNT (\$83/\$84) will contain the address of the beginning of the variable’s data field.

PTRGET can be used to simplify the passing of Applesoft variables to and from an assembly-language program. Variables are usually passed by tacking their names, separated by commas, on to a CALL or & command. For example, to pass two variables, say F% and L%, to a program starting at \$300, you could use the following command:

```
CALL 768,F%,L%
```

Immediately after the CALL 768 command is executed, TXTPTR is pointing to the location occupied by the comma separator. Before we can use PTRGET, TXTPTR must be advanced by one position. This is done at the beginning of the subroutine being called by using a subroutine called CHKCOM (\$DEBE) that ensures that the current character is indeed a comma, and then increments TXTPTR. Once this has been done, everything is ready for a call to PTRGET. After it has been called, VARPNT (\$83/\$84) can be examined to determine where the data for that variable is located. Since the storage format of the data is known, it is a simple matter to read and interpret it. In summary, the method to be used to locate a variable is as follows:

```
JSR CHKCOM      ;Skip over the "," separator
JSR PTRGET      ;Find the variable and put ptr
                  in VARPNT
LDY #0
LDA (VARPNT),Y ;Get first byte of variable's
                  data
.
INY
LDA (VARPNT),Y ;Get second byte of variable's
                  data
```

After PTRGET has been called, TXTPTR points to the byte after the last character of the variable's name. This means that you would perform another

```
JSR CHKCOM
JSR PTRGET
```

sequence to retrieve the next variable specified after the CALL statement.

Let's look at a complete example to see how to use these subroutines. Table 4-10 shows a program called UPPER that is designed to convert any lowercase characters in a string variable to their corresponding uppercase characters. Once the program has been installed, it can be called from Applesoft as follows:

```
CALL 768,A$
```

where A\$ is the string variable to be modified.

Notice how the program works. The first step is to skip over the comma by calling CHKCOM. After that has been done, TXTPTR will be pointing to the

Table 4-10. UPPER, a program to convert the lower-case characters in a string variable to upper-case.

```

1 00000000 *****
2 00000000 *
3 00000000 *
4 00000000 *
5 00000000 *
6 00000000 *
7 00000000 *
8 00000000 *
9 00000000 *
10 00000000 *
11 00000000 *
12 00000000 *
13 00000000 *
14 00000000 *
15 00000000 *
16 00000000 *
17 00000000 *
18 00000000 *
19 00000000 *
20 00000000 *
21 00000000 *
22 00000000 *

```


0310:	C8	23	INY			
0311:	B1 83	24	LDA	(VARPNT),Y ;Get string pointer (high)		
0313:	85 07	25	STA	STRING+1		
		26				
0315:	8A	27	TXA			
0316:	A8	28	TAY			
0317:	C0 00	29	CPY	#0		
0319:	F0 16	30	BEQ	ALLDONE		
031B:	88	31	DEY			
031C:	C0 FF	32	CPY	#\$FF		
031E:	F0 11	33	BEQ	ALLDONE		
0320:	B1 06	34	LDA	(STRING),Y ;Get string element		
0322:	C9 61	35	CMP	#'a		
0324:	90 F5	36	BCC	SCAN		
0326:	C9 7B	37	CMP	#'z+1		
0328:	B0 F1	38	BCS	SCAN		
032A:	29 DF	39	AND	#\$DF		
032C:	91 06	40	STA	(STRING),Y ; and put it into string.		
032E:	4C 1B	41	JMP	SCAN		
0331:	60	42	RTS			

					ALLDONE	
--	--	--	--	--	---------	--

						03
--	--	--	--	--	--	----

						03
--	--	--	--	--	--	----

“A” in A\$ and PTRGET can be called to locate the three data bytes used to describe the string (one byte for the length and two bytes representing its location). The pointer to the first of these three bytes is automatically stored in VARPNT (\$83/\$84), so that the three bytes can be examined by using indirect indexed instructions as follows:

```
LDA (VARPNT),Y
```

where Y=0,1,2. After the length and pointer have been determined, it is a simple task to scan through the bytes in the string to see whether their ASCII codes are between those for “a” and “z” and, if they are, to convert them to uppercase by performing an AND #\$DF operation. (This essentially subtracts 32 from the lowercase ASCII code, thus converting it to the corresponding uppercase ASCII code.) If you print A\$ after calling UPPER, you will see that all of its lowercase characters have, indeed, been converted to uppercase.

Evaluating Formulas

Not surprisingly, there are also several built-in Applesoft subroutines that can be used to evaluate mathematical formulas. Again, you could write such programs yourself, but they would need to be exceedingly complex and would be difficult to develop.

The main Applesoft subroutine for evaluating a mathematical formula is called FRMNUM and is located at \$DD67. To use it, you must first ensure that TXTPTR is, as usual, pointing to the location of the first character in the formula. Once this has been done, FRMNUM can be called; after this subroutine has finished executing, the result of the calculation will be stored in the FAC. You can then use other built-in subroutines to massage this number as you see fit, for example, to print it out or to convert it to an integer.

Let's look at an example of the use of FRMNUM. The program in Table 4-11, called FORMULA, evaluates any mathematical formula that is passed to it and displays the result. To CALL it from Applesoft, you must enter the command

```
CALL 768,aexpr
```

where “aexpr” represents the Applesoft formula that is to be evaluated.

The first part of the program should look familiar. It is the “standard” JSR CHKCOM instruction that skips over the comma after the CALL statement. Once this has been performed, the formula can be evaluated by a JSR FRMNUM and the result will be placed in the primary FAC. To see the result it is simply necessary to perform a JSR PRNTFAC (\$ED2E).

Converting Numbers

Number conversion plays an important role in the Applesoft interpreter. Numbers are normally handled internally in a binary format, but whenever

Table 4-11. FORMULA—a program to demonstrate the use of Applesoft's formula evaluation subroutines.

```

1  *****
2  *          FORMULA
3  *          *
4  *      CALL 768,[formula]
5  *          *
6  *      This program evaluates
7  *      and displays an Applesoft
8  *      mathematical formula.
9  *      *****
10 *****
11 FRMNUM EQU $DD67      ;Evaluate formula
12 CHKCOM EQU $DEBE      ;Check for comma and move on
13 PRINTFAC EQU $ED2E    ;Display the result
14
15 ORG $300
16
17 JSR CHKCOM      ;Skip over comma separator
18
19 *****
20 * Evaluate the formula and put *
21 * it in the FAC.
22 *****
23 JSR FRMNUM
24
25 *****
26 * Convert the number in FAC *
27 * to decimal and display it.*
28 *****
29 JSR PRINTFAC
30
31 RTS

```

0300: 20 BE DE

0303: 20 67 DD

0306: 20 2E ED

0309: 60

they are to be displayed they must be converted to more recognizable decimal numbers. Conversely, numbers that are inputted, say from the keyboard by a user, are normally inputted in decimal form and must be converted to binary form before they can be processed.

In addition to the above types of conversions, it is often necessary to convert an integer number to a floating-point number and vice versa. This is handled by the Applesoft GIVAYF subroutine and by the CONINT or GETADR subroutines, respectively. The latter two subroutines are especially useful because quite often only integer quantities are being manipulated and, as we have seen, whenever a formula is evaluated by performing a JSR FRMNUM, the result is placed in the FAC, which is difficult to interpret. By using CONINT or GETADR, the FAC can be quickly converted to an easy-to-handle one- or two-byte integer format.

The program in Table 4-12, CONVERT, shows how the CONINT subroutine can be used to convert the contents of the FAC to a one-byte integer in the range 0 . . . 255. As usual, CONVERT is designed to be called from Applesoft, using the command

```
CALL 768,aexpr
```

where "aexpr" represents a mathematical formula that will evaluate into an integer within the 0 . . . 255 range.

The first step is to skip over the comma with a JSR CHKCOM. Then, the formula is evaluated and placed in the primary FAC with a JSR FRMNUM. At this stage, we would like to convert the FAC into an easier format to handle: a one-byte number. This is done by executing a JSR CONINT; after CONINT has been executed, the one-byte number will be found in the X register. CONVERT then stores the value in the X register at location \$6 where it can be read with an Applesoft PEEK (6) command.

If the integer result is going to be larger than 255, then GETADR must be used instead of CONINT. After a JSR GETADR, the value of the integer will be contained in LINNUM (\$50/\$51), so that the decimal result will be $\text{PEEK}(80) + 256 * \text{PEEK}(81)$.

Table 4-12. CONVERT—a program to demonstrate the use of Applesoft's number conversion subroutines.

```

1 ***** CONVERT *****
2 *
3 *
4 * CALL 768,[formula] *
5 *
6 *****
7 RESULT EQU $6 ;Store the answer here
8
9 CHKCOM EQU $DEBE ;Check for comma and move on
10 FRMNUM EQU $DD67 ;Evaluate a formula
11 CONINT EQU $E6FB ;Convert FAC to integer
12
13 ORG $300
14
15 JSR CHKCOM ;Skip over comma
16 JSR FRMNUM ;Evaluate the formula
17 JSR CONINT ;Put one-byte answer in X
18
19 STX RESULT ;Store the answer
20 RTS

```

Further Reading for Chapter 4

Standard reference works . . .

Applesoft BASIC Programmer's Reference Manual, Volumes 1 and 2, Apple Computer, Inc., 1982.

All About Applesoft, Call -A.P.P.L.E., 1981. A useful source of internal information about Applesoft.

BASIC Programming with ProDOS, Apple Computer, Inc., 1983.

On Applesoft entry points . . .

J. Crossley, "Applesoft Internal Entry Points", *Apple Orchard*, March/April 1980. The seminal work on Applesoft entry points. Unfortunately, it contains numerous typographical errors and incorrect addresses—these corrections have been made in a reprint of the article which appears in "All About Applesoft", above.

R.M. Mottola, "Applesoft Floating Point Routines", *Micro*, August 1980, p. 53. A detailed look at Applesoft's built-in subroutines that support real-number mathematics.

C. Bongers, "In the Heart of Applesoft", *Micro*, February 1981, p. 31. A comprehensive look at the internals of the Applesoft interpreter.

"Using Applesoft ROMs from Assembly Language", *Apple Assembly Line*, November 1981, pp. 2–13. More on accessing Applesoft's built-in subroutines.

C. Bongers, Applesoft's CHARGET Routine, *Call -A.P.P.L.E.*, March 1982, p. 21. Suggestions for improvements to CHARGET.

B. Sander-Cederlof, "All About PTRGET & GETARYPT", *Apple Assembly Line*, March 1983, pp. 2–9. A look at two useful entry points to Applesoft.

On Applesoft data storage . . .

V. Golding, "Applesoft From Bottom to Top", *Call -A.P.P.L.E.*, March 1979, p.3. A look at the internal structure of Applesoft. A revised version of this article appears in "All About Applesoft", above.

G.A. Lyle, "Float, Float, Float Your Point (F.P. Representation)", *Apple Orchard*, Winter 1980, pp. 37–39. A description of how Applesoft stores real variables.

E.E. Goetz, "Real Variable Study", *Call -A.P.P.L.E.*, January 1981, pp. 8–23. A detailed look at how Applesoft deals with real variables. A revised version of this article appears in "All About Applesoft", above.

C.K. Mesztenyi, "Applesoft Internal Structure", *Call -A.P.P.L.E.*, January 1982, p.9.

A. Moss, "Playing With Program Pointers", *Nibble*, Vol. 4, No. 3 (1983), pp. 69–81. A look at the various Applesoft pointers.

On linking to assembler language . . .

B. Sander-Cederlof, "Using USR for a WEEK", *Apple Assembly Line*, October 1982, p. 30. A program is presented which uses USR to calculate the value of a two-byte pointer.

D. Lingwood, "The Return of the Mysterious Mr. Ampersand", *Call -A.P.P.L.E.*, May 1980, p.26. Examples of uses for the & command.

Source Code for the Applesoft interpreter . . .

Available from:

- (1) Roger Wagner Publishing, P.O. Box 582, Santee, CA 92071
(comes with and requires Merlin Assembler).
- (2) S-C Software Corporation, P.O. Box 280300, Dallas, TX 75228
(requires S-C Macro Assembler).

5

The ProDOS Disk Operating System

The disk drive is the primary mass storage device used by the //c. It can be used to quickly and reliably access units of information (called “files”) that are stored on standard 5.25-inch floppy diskettes. These files can contain programs, data, readable text, or any other type of information.

Information is passed to and from a diskette by using special disk operating system (DOS) commands that are available for use by an Applesoft program after a DOS diskette is first started up (“booted”). Assembly-language programs can access the diskette by calling documented DOS subroutines. The DOS that comes with the //c is called ProDOS and we will be examining it in detail in this chapter.

ProDOS was first released by Apple in January 1984 for use with its Apple II family of computers. ProDOS is intended to be the successor to Apple’s older disk operating system (called DOS 3.3). DOS 3.3 is essentially the same operating system that has been in use since Apple first introduced its disk drive peripheral for the original Apple II in 1978.

The //c comes with a built-in disk drive that is functionally identical to the Apple Disk II drives that are used on the Apple //e, Apple II Plus, and Apple II. A second disk drive can also be added by attaching it to the special disk port at the back of the //c.

The built-in disk drive is the //c’s port 6 device. The diskette in this drive will automatically boot when the power to the //c is turned on. Alternatively, you can boot it by entering a “PR#6” command from Applesoft direct mode, a “6[control-P]” command from the system monitor, or by pressing [control-OPEN- APPLE-RESET] at any time.

The external drive is sometimes interpreted as a port 7 device, but ProDOS, for DOS 3.3 compatibility reasons, considers it to be drive 2 of the port 6 device. This means that the external drive cannot be booted using a “PR#7” command, as you might otherwise expect. However, you can use the system monitor “7[control-P]” command to boot it.

The primary purpose of this chapter is not to teach you how to use the

ProDOS commands but rather to explain the methods used by ProDOS to organize information on diskettes and to provide you with an insight into the internal operation of ProDOS. Let's get started right now.

Formatting Diskettes

Before a diskette can be used by ProDOS it must be formatted into a state that ProDOS recognizes. (To do this, select the "FORMAT A DISK" command in the program found on Apple's System Utilities diskette for the //c.) When you format a diskette, templates for 35 "tracks" on the diskette are created (numbered from 0 to 34), each of which can hold 4096 bytes of information. These tracks are arranged in concentric rings around the central hub of the disk, with track 0 being located at the outside edge and track 34 at the inside edge. ProDOS can access any track by causing a special read/write head (located inside the disk drive) to move over the desired track. This is done using I/O locations that activate a small motor (called a "stepping" motor) that controls the motion of a metal arm to which the read/write head is connected. This arm moves along a radial path beginning at the outside edge of the diskette (track 0) and ending at the inside edge (track 34).

Each of the 35 tracks that are formatted on a diskette are subdivided into sixteen smaller units called "sectors." The sectors that make up a track are numbered from 0 to 15 and each can hold exactly 256 bytes of information. If you do the mathematics, you will quickly determine that a diskette can hold 560 sectors (140K) of information.

Note, however, that when ProDOS organizes files on a diskette it uses the 512-byte block as the basic unit of file storage; each block is made up of two sectors. An initialized diskette is considered to be made up of 280 blocks (numbered from 0 . . . 279), but it is rarely necessary to know where these blocks are actually located on the diskette since ProDOS assigns block numbers to physical locations internally.

ProDOS Memory Map

When a ProDOS diskette is first booted, a file called PRODOS is loaded into memory and executed (it is a ProDOS "system" file). This file contains the fundamental I/O subroutines that are used to read and write blocks of data from and to the diskette. PRODOS then loads and executes another system file into memory; the one loaded has a name of the form xxxx.SYSTEM (the first file having such a name when the disk is catalogued will be used).

If an Applesoft programming environment is to be supported, this file must be BASIC.SYSTEM (it is found on the ProDOS system diskette). As we will see below, BASIC.SYSTEM contains the subroutines that "add" the standard ProDOS commands to the Applesoft programming language. It also takes care

of parsing these commands, doing syntax checking, and calling the PRODOS I/O subroutines when required. For convenience, we will be referring to the resultant PRODOS-BASIC.SYSTEM program combination as “ProDOS” even though this is technically not the case.

After ProDOS has been loaded as described, it will occupy the following memory locations:

- \$9A00-\$BFFF in main RAM
- \$D000-\$DFFF in bank1 of main bank-switched RAM and \$D100- to \$D3FF in bank2 of main bank-switched RAM
- \$E000-\$FFFF in main bank-switched RAM

(See Chapter 8 for a discussion of bank-switched RAM.) In addition, a general-purpose file buffer will be set up from \$9600 to \$99FF and the Applesoft HIMEM location will be set equal to \$9600. (HIMEM refers to the value of the Applesoft end-of-string pointer at \$73/\$74—see Chapter 4.) Thus, the area used for storage of Applesoft string variables will not conflict with areas used by ProDOS.

The \$400-byte buffer just above Applesoft HIMEM is always used by ProDOS as a buffer for directory blocks whenever the diskette is CATALOGued. This buffer does not always begin at \$9600, however, since HIMEM could be changed in the following instances:

- By using the Applesoft HIMEM: command or the GETBUFR (\$BEF5) subroutine in BASIC.SYSTEM
- By opening and closing diskette files using the ProDOS OPEN and CLOSE commands

The Applesoft HIMEM: command simply places the address specified after the command directly into the HIMEM pointer.

The GETBUFR (\$BEF5) subroutine can be called from an assembly-language program if you want to lower HIMEM by a given number of 256-byte pages. To do this, call GETBUFR with the number of pages in the accumulator; upon exit, the carry flag will be clear if there was enough free space to lower HIMEM, or it will be set if there wasn't.

It is not immediately obvious how or why the OPEN and CLOSE commands affect the value of HIMEM. Whenever a file is opened, ProDOS creates a \$400-byte file buffer by moving HIMEM down in memory by that number of bytes and then reserving the \$400 byte area beginning at the original HIMEM position for use by the file. Whenever a file is closed, HIMEM is moved up by \$400 bytes. While doing all this, ProDOS takes all steps necessary to ensure that Applesoft's string variables are not overwritten.

It is often convenient to reserve a safe area of memory where assembly-language programs can be stored without fear of being overwritten by either ProDOS or Applesoft. In DOS 3.3 days, such an area could be reserved simply by lowering HIMEM and then storing the program between the new and old

HIMEM addresses. You can't do this with ProDOS, however, because of the way HIMEM changes when files are opened or closed. When ProDOS is being used, however, a safe area can be reserved that resides just above the \$400-byte directory buffer that sits above HIMEM. The steps that must be followed to do this are as follows:

- Close all files using the ProDOS CLOSE command.
- Lower HIMEM by a multiple of \$100 (256) bytes using the Applesoft HIMEM: command or the GETBUFR (\$BEF5) subroutine.

If you are using the HIMEM: command to lower HIMEM, these steps must be performed before any Applesoft variables have been defined, since the Applesoft string space will be overwritten. After these two steps have been completed, the area from HIMEM + \$400 to \$99FF can be used for storage of machine-language programs without danger of having them overwritten by ProDOS operations.

Keep in mind one important restriction that applies when using ProDOS: if HIMEM is being changed (that is, the \$73/\$74 end-of-string pointer is being changed), it must be changed in multiples of 256 bytes only! You can't go wrong if you use GETBUFR, of course, since it can only be used to lower HIMEM by page multiples. Be very careful, however, when you use the Applesoft HIMEM: command, because Applesoft does not check to see that the address specified in the command is an integral multiple of 256.

ProDOS Page 3 Vectors

Apple has reserved the entire area from \$3D0 . . . \$3EF for use by ProDOS even though the current version (1.1) uses only the first six locations. As indicated in Table 5-1, these six locations hold two JMP instructions to the warm-start entry point of ProDOS (location \$BE00).

ProDOS also initializes all of the system vectors that appear in page 3 from \$3F0 . . . \$3FF. These are the interrupt vectors for BRK, Reset, IRQ, and NMI (see Chapter 2), the vector for the system monitor's [control-Y] USER command (see Chapter 3), and the vector for the Applesoft & command (see Chapter 4). Descriptions of all the vector subroutines installed by ProDOS are given in Table 5-2.

Filenames and Pathnames

When a file is stored on a diskette, it is assigned a unique filename that is used to identify it thereafter. A ProDOS filename can be up to 15 characters long. It must begin with an alphabetic letter (A–Z), but the other characters may be any combination of letters, digits (0–9), and periods (.). Lowercase

Table 5-1. ProDOS page 3 vectors.

<i>Address</i>	<i>Description of Vector</i>
\$3D0-\$3D2	A JMP instruction to the ProDOS warm-start entry point. A call to this vector will reconnect DOS without destroying the Applesoft program in memory. Use the "3D0G" command to move from the system monitor to Applesoft.
\$3D3-\$3D5	Another JMP instruction to the ProDOS warm-start entry point.

letters may also be used, but ProDOS automatically converts them to uppercase. Here are some examples of valid ProDOS filenames:

```
CRACKED.WHEAT
CANNED.HEAT
MY.PROGRAM
```

When a file is saved to diskette, it can be stored in any one of several "directories" that can be created on the diskette. These directories are somewhat analogous to file folders in that they are usually used to hold groups of related files. For example, you may use one directory to hold word processing documents, another to hold Applesoft programs, and so on. The ability to create separate directories on a diskette makes it easier to deal with large numbers of files.

When a diskette is first formatted, only one directory, called the "volume" directory, exists; you name the volume directory when the diskette is formatted (the rules for naming directories are the same as for naming files). You can create additional directories (called "subdirectories") within the volume directory using the ProDOS CREATE command. Indeed, you can even create subdirectories within subdirectories (up to 64 levels are permitted).

The directory in which a file is to be saved is normally specified by tacking on a special prefix to the filename to create a unique identifier called a "pathname." A pathname is made up of the names of a series of directories, beginning with the name of the volume directory and continuing with the names of all the directories that must be passed through to reach the directory you want, followed by the filename itself; each directory name is separated from the next by a slash ("/") and a slash must precede the name of the volume directory. The list of directory names must define a continuous path: each directory specified must be contained within the preceding directory.

For example, consider a diskette that has a volume directory called BASEBALL and two subdirectories within BASEBALL called AMERICAN and NATIONAL. If you want to save a file called NY.YANKEES in the AMERICAN subdirectory, then you would specify the following pathname:

```
/BASEBALL/AMERICAN/NY.YANKEES
```

Table 5-2. Initialization of page 3 system vectors by ProDOS.

Vector Name	Address	Contents	Description
BRK	\$3F0-\$3F1	\$FA59	Address of a subroutine to that displays the 65C02 registers and then enters the system monitor.
RESET	\$3F2-\$3F3	\$BE00	Address of the ProDOS warm-start entry point (reconnects ProDOS).
&	\$3F5-\$3F7	"JMP \$BE03"	Jump to ProDOS's external entry point for command strings (see Apple's ProDOS Technical Reference Manual).
USER	\$3F8-\$3FA	"JMP \$BE00"	Jump to ProDOS's warm-start entry point.
NMI	\$3FB-\$3FD	"JMP \$FF59"	Jump to the system monitor's cold-start entry point.
IRQ	\$3FE-\$3FF	\$BFEB	Address of the special ProDOS interrupt handler (see Chapter 2).

Note: The addresses stored at each vector location are stored with the low-order byte first.

If you simply specified NY.YANKEES, then the file would be saved in the currently active directory, which is usually the volume directory (unless it has been changed using the PREFIX command that we are about to describe).

If all files of interest are contained in the same subdirectory, it becomes annoying to have to specify the same chain of directory names leading up to the file every time one is to be used. To alleviate this annoyance, ProDOS supports a PREFIX command that can be used to set the chain of directory names to which any filename specified in a ProDOS command will be automatically appended. For example, if PREFIX is set by entering the following ProDOS command:

```
P R E F I X / B A S E B A L L / A M E R I C A N /
```

then any file contained in the directory at the end of this path (AMERICAN) can be referred to by entering its filename only. A name that is a continuation of the prefix could also be entered to access files in lower-level subdirectories; such a name is called a "partial pathname." If PREFIX is set as just described and if AMERICAN contains a subdirectory called CHAMPS which contains a

file called TIGERS.1984, then you would access the file by specifying a partial pathname of CHAMPS/TIGERS.1984.

The advantage of subdirectories is often not readily apparent to users of floppy diskettes, but becomes obvious when a hard disk system is used where there is enough room to hold thousands of files. If all the files were held in one directory you might have to wait a long time to spot your file when the disk was catalogued, and even then you could well miss it amidst the multitude of other files. Fortunately, the hierarchical directory structure provided by ProDOS allows related files to be grouped within the same subdirectory for easy access.

By the way, when ProDOS is first booted, it treats the 64K of auxiliary memory as if it was storage space on a diskette and assigns it the volume directory name of /RAM. A total of 119 blocks on this volume are available for file data storage. The /RAM volume can be used to load and save programs extremely quickly since "I/O" operations involve nothing more than the movement of data between memory locations; no slow mechanical parts need be controlled. Remember, however, that any information stored in the /RAM volume will disappear as soon as the //c is turned off or when ProDOS is rebooted. If you want to permanently save any files on the /RAM volume, you must transfer them to a diskette first.

BASIC.SYSTEM Commands

After ProDOS (version 1.1) is booted and the BASIC.SYSTEM system program is executed, 31 commands become available for use within an Applesoft program (many can also be used from Applesoft direct mode). Most of these commands provide ready access to files for I/O operations (OPEN, READ, POSITION, WRITE, APPEND, FLUSH, and CLOSE), or general file management (CAT, CATALOG, CREATE, DELETE, LOCK, PREFIX, RENAME, UNLOCK, and VERIFY), or program file loading and execution (-, BLOAD, BRUN, BSAVE, EXEC, LOAD, RUN, SAVE). There are also commands used to effect I/O redirection (IN#, PR#), to perform garbage collection of Applesoft variables (FRE), to save and load Applesoft variables to and from files (STORE and RESTORE), to transfer control from one Applesoft programs to another without destroying existing variables (CHAIN), and to disconnect BASIC.SYSTEM and run a ProDOS system program (BYE).

To use an Applesoft command from within a program, you must use the PRINT statement to print a [control-D] character, the ProDOS command, the command arguments, and then a carriage return. For example, to list all the files in the //c's /RAM volume, you would execute a line that looks something like this:

```
100 PRINT CHR$(4);"CATALOG/RAM"
```

In this example, the CHR\$(4) statement generates the [control-D] character,

the ProDOS command is CATALOG, and the command argument is /RAM. The required carriage return is automatically generated by the PRINT statement.

If you're entering a ProDOS command directly from the keyboard in Applesoft direct mode, then you don't have to worry about the [control-D]. All that you have to do is type in the command followed by the command arguments. The keyboard equivalent of the CATALOG command is simply

CATALOG/RAM

You should be aware, however, that ProDOS does not permit all of its commands to be entered from the keyboard in this way.

Most of the ProDOS commands accept an argument that is simply the filename, pathname, or partial pathname of the file that is to be acted on. The major exceptions are the PR#, IN#, and FRE commands.

Let's take a quick look at each of the 31 ProDOS commands right now. (You can refer to Apple's manual, "BASIC Programming with ProDOS", for detailed information on these commands.) They can be divided into four distinct categories: file management commands, file loading and execution commands, file input/output commands, and miscellaneous commands.

File Management Commands

CAT. This command is used to display a list of the names of the files on the diskette. Only the names of the files in the directory specified in the argument following the CAT command will be displayed (if no argument is specified, then the currently active directory will be used). CAT will also display the type of each file (as a three-character mnemonic such as BAS, BIN, TXT, SYS, and so on; see Table 5-6), the number of blocks it occupies, and the date on which it was last modified.

CATALOG. This command is very similar to CAT. It displays the very same information for each file as well as its time of last modification, its creation date and time, its size (in bytes), and its "subtype" entry (which is either the default loading address for a binary file or the record length for a textfile).

CREATE. This command is used to create a file on the diskette. It is most often used to create subdirectory files since the other common types of ProDOS files (Applesoft programs, binary files, and text files) are automatically created by other ProDOS commands (SAVE, BSAVE, and OPEN). For example, if the volume directory is active and you want to create a subdirectory called DEMO.PROGRAMS, then you would enter the command

CREATE DEMO.PROGRAMS

from the keyboard. When you do this, the subdirectory will appear as a file entry when you catalog the directory in which it is contained. The file type mnemonic used to identify it in the catalog listing is DIR.

DELETE. This command is used to erase a file from the diskette. Only unlocked files can be erased with the DELETE command. (See the description of the LOCK command.)

LOCK. This command is used to protect a ProDOS file from accidental deletion or modification. Once a file has been locked, it cannot be deleted or modified unless it is first unlocked. You can tell which files on a diskette are locked by cataloguing the diskette (using the CAT or CATALOG commands); if the name of the file is preceded by an asterisk ("*"), it is locked.

PREFIX. This command is used to define the chain of directory names to which any filename or partial pathname specified will automatically be appended to generate the full pathname (see above). It is this pathname on which the ProDOS commands will act.

RENAME. This command is used to change the name of any file on the diskette.

UNLOCK. This command unlocks a file that is locked.

VERIFY. This command checks to see if a file exists on the diskette. If it doesn't, an error message will be displayed.

File Loading and Execution Commands

- (dash). This is the ProDOS "intelligent" run command. Its argument can be the name of an Applesoft program, a binary program, or a textfile, in which cases the - will behave exactly like a RUN, BRUN, or EXEC command, respectively.

BLOAD. This command is used to transfer the data from a file to a contiguous block of memory. The most common form of this command is:

```
BLOAD MY.FILE, Aaddr
```

where "addr" is the address of the beginning of the block to which the file is to be transferred. "addr" can be a decimal or hexadecimal number; as usual, hexadecimal addresses are preceded by "\$." The BLOAD command can also be used without the ",Aaddr" suffix; in this case, the file will be loaded at the location from which it was originally saved to diskette using the BSAVE command (this address appears in the "subtype" column when the diskette is catalogued using the CATALOG command).

BRUN. This command is the same as BLOAD except that after the file is loaded, it will automatically be executed. Execution begins at the loading address.

BSAVE. This command is used to save the contents of a range of memory to a file. (The default file type used is binary (BIN) but you can override this

default.) For example, to save the contents of memory from \$300 to \$3CF to a binary file called PAGE.THREE, you would enter the command:

```
BSAVE PAGE .THREE ,A$300 ,E$3CF
```

or

```
BSAVE PAGE .THREE ,A$300 ,L$D0
```

where the “,A\$300” suffix indicates the starting address of the range, “,E\$3CF” indicates the ending address, and “,L\$D0” indicates the number of bytes to be saved. You can also use decimal numbers to specify addresses and numbers.

EXEC. This command is used to redirect subsequent requests for input to the specified file instead of the keyboard until everything in the file has been read. For example, suppose you have defined a file called MY.STARTUP that contains the following two lines:

```
HOME  
CATALOG
```

When you enter EXEC MY.STARTUP from direct mode, the screen will clear and then the diskette will be catalogued, just as if you had entered the two commands directly from the keyboard.

LOAD. This command is used to load an Applesoft program into memory from a file.

RUN. This command is the same as the LOAD command except that after the program is loaded, it will automatically be executed.

SAVE. This command is used to save an Applesoft program to a file. The file type mnemonic for a program file is BAS.

File Input/Output Commands

OPEN. This command is used to open a file, usually a textfile, for either reading or writing. If the filename specified does not exist, then a new file will be created. A file must be opened before it can be accessed using the ProDOS READ, WRITE, APPEND, FLUSH, and POSITION commands. Textfiles can be opened as one of two basic types: sequential or random-access. A sequential textfile is one in which lines of information are stored one after another, with no gaps in between; usually, if you want to access information anywhere in the file, you have to read all the information that precedes it.

A random-access file is one that is organized as a series of fixed-length records that hold related groups of information; any record can be accessed “randomly” (that is, without reading all previous records first) simply by specifying its record number when using the READ command. The record length is assigned to a random-access textfile when it is first created or opened; it is displayed in the subtype column of a CATALOG listing in the form

"R=\$xxxx." For example, if the record length is 127, then the subtype entry would be "R=\$007F."

READ. This command is used to redirect subsequent requests for input to an open file instead of the keyboard.

POSITION. This command is used to adjust the position in the file at which subsequent read and write operations will operate.

WRITE. This command is used to redirect subsequent output to an open file instead of to the video screen.

APPEND. This command is used to open a file and to redirect subsequent output to the end of it.

FLUSH. When a ProDOS file is opened, a 512-byte file buffer is allocated to it in memory. Data that is written to the file is not normally stored to disk until this buffer becomes full; when it does, the buffer is saved to disk and then it is cleared. The FLUSH command is used to force any data stored in the buffer to be saved to diskette even if the buffer is not yet full. This minimizes the risk of data loss in the event of an unexpected exit from the program (caused by a loss of power, pressing RESET, and so on) but it slows down diskette write operations considerably.

CLOSE. This command is used to close a file that has been opened using the OPEN command.

Miscellaneous Commands

BYE. This command is used to disconnect BASIC.SYSTEM and execute a ProDOS system program. When the command is entered, you will be prompted to enter the prefix and partial pathname of the system program. After you do this, the program will be executed.

CHAIN. This command is used to transfer control from one Applesoft program to another while maintaining the names and current values of all the variables in the program from which control is being passed. This allows very large programs to be executed on the //c by breaking them into separate modules and chaining them together.

FRE. This command is used to force garbage collection of Applesoft string variables. (See Chapter 4 for a description of this procedure.) This garbage collection command is much faster than the one built in to the Applesoft interpreter.

IN#. This command is used to redirect subsequent requests for input to a port or to a specified memory location. (See Chapter 6.)

PR#. This command is used to redirect subsequent output to a port or to a specified memory location. (See Chapter 7.)

RESTORE. This command is used to initialize the names and values of the variables in an Applesoft program to those contained in the file specified in the argument. This file must be of type VAR (this is the type created by the STORE command).

STORE. This command is used to save the names and current values of all the variables in an Applesoft program to a file. The mnemonic for the file type code assigned to the file is VAR.

ProDOS File Storage

We are now ready to examine the method that ProDOS uses to store information on a diskette. This will include an analysis of the structures of the directories that hold information concerning files, of the volume bit map that keeps track of block usage on the diskette, and of the index blocks that are used to locate the data blocks that are used by each file. We'll also see how to read and write blocks on the diskette using internal ProDOS subroutines.

Before we continue, keep in mind that the descriptions that follow relate to ProDOS only and not to its predecessor, DOS 3.3.; the older DOS 3.3 organizes files on a diskette in a less efficient, and wholly incompatible, manner. A utility program called CONVERT is included with ProDOS, however, that allows most files to be transferred between DOS 3.3 and ProDOS formatted diskettes so that they can be used by either operating system.

Volume Bit Map

Of the 280 blocks on a ProDOS diskette, the first seven (numbered from 0 to 6) are reserved for specific purposes. Blocks 0 and 1 contain a program that is loaded into memory whenever the diskette is booted. This program is called the bootstrap loader and is responsible for loading and executing the PRODOS system file. Blocks 2 through 5 represent the four blocks that contain the volume directory information and will be described in the next section. Block 6 contains the volume bit map for the diskette.

The volume bit map is used to keep track of which areas of the diskette are in use and which are free. Only the first 35 bytes (280 bits) in the volume bit map block are actually used and each bit in each byte corresponds to a unique block number. The byte number (from 0 to 34), and the bit number within that byte (from 0 to 7), that corresponds to any given block number (from 0 to 279) can be calculated using the following Applesoft formulas:

```
BYTENUM = INT(BLOCKNUM/8)
BITNUM = 7 - BLOCKNUM - 8 * BYTENUM
```

If the bit associated with a particular block is one, then that block is free. If it is zero, then it is being used by a file on the diskette.

Diskette Directory

As was explained earlier, ProDOS allows multiple directories to be created on one diskette. With the exception of the volume directory (the one through which all the others must be accessed), these directories can be stored just about anywhere on the diskette since they are treated similarly to standard files. The volume directory, however, is always located in blocks 2 through 5 of the diskette.

Each block used by any directory can hold up to thirteen 39-byte file entries. (This means that the four-block volume directory can hold a total of 52 entries, one of which is the volume name entry.) These entries completely describe the files by specifying the name, type, and size of the file. The map of a directory block is shown in Table 5-3.

The first block used by a directory (or subdirectory) is called the “key block” and is configured slightly differently than the others. The 39-byte entry that normally describes the first file in the block is instead used to describe the directory itself. This entry is called the directory header.

Table 5-3. Map of a ProDOS directory block.

<i>Byte number in Directory Block</i>	<i>Meaning of Entry</i>
\$000-\$001	Block number of the previous directory block (low byte first). This will be zero if this is the first directory block.
\$002-\$003	Block number of the next directory block (low byte first). This will be zero if this is the last directory block.
\$004-\$02A	Directory entry for file #1 OR, if this is the first block of the directory (bytes \$00 and \$01 are 0), the directory header.
\$02B-\$051	Directory entry for file #2
\$052-\$078	Directory entry for file #3
\$079-\$09F	Directory entry for file #4
\$0A0-\$0C6	Directory entry for file #5
\$0C7-\$0ED	Directory entry for file #6
\$0EE-\$114	Directory entry for file #7
\$115-\$13B	Directory entry for file #8
\$13C-\$162	Directory entry for file #9
\$163-\$189	Directory entry for file #10
\$18A-\$1B0	Directory entry for file #11
\$1B1-\$1D7	Directory entry for file #12
\$1D8-\$1FE	Directory entry for file #13
\$1FF	<Not used>

The meaning of each of the 39 bytes that make up a directory header are shown in Table 5-4. Notice the differences between the header for a volume directory and the header for a subdirectory.

Table 5-4. Map of a ProDOS directory header.

Byte number in Key Block	Description (usual entries in parentheses)
\$04	High four bits: storage type —\$0F for a volume directory —\$0E for a subdirectory Low four bits : length of directory name
\$05–\$13	Directory name (in ASCII). The length of the name is contained in the low-order half of byte \$04.
\$14–\$1B	<Reserved>
\$1C–\$1D	The date on which this directory was created (format: MMDDDDDD YYYYYYYM)
\$1E–\$1F	The minute (byte \$1E) and hour (byte \$1F) at which this directory entry was created.
\$20	The version number of ProDOS that created this directory.
\$21	The lowest version of ProDOS that is capable of using this directory.
\$22	Access code for this directory (see Figure 5-1)
\$23	The number of bytes occupied by each directory entry (39).
\$24	The number of directory entries that can be stored on each block (13).
\$25–\$26	The number of active files in this directory (not including the directory header)
\$27–\$28	VOLUME DIRECTORY: The block where the volume bit map is located (6) SUBDIRECTORY: the block in which the entry defining this subdirectory is located (this is in the parent directory of the subdirectory).
\$29–\$2A	VOLUME DIRECTORY: The size of the volume in blocks (280).
\$29	SUBDIRECTORY: The directory entry number within the block given by \$27/\$28 that defines this subdirectory (1 to 13).
\$2A	SUBDIRECTORY: The number of bytes in each directory entry of the parent directory (39).

All directory entries that do not represent directory headers represent either standard data files (for example, binary files, text files, and Applesoft programs) or subdirectory files. The formats of the directory entries for both of these two types of files are virtually identical and are as shown in Table 5-5.

Table 5-5. Map of a ProDOS directory file entry.

<i>Relative Byte Number</i>	<i>Meaning of Entry</i>
\$00	High four bits: storage type (see text) —\$00 for an inactive file —\$01 for a seedling file —\$02 for a sapling file —\$03 for a tree file —\$0D for a subdirectory file Low four bits : length of file name
\$01–\$0F	File name (in ASCII with bit 7 = 0)
\$10	File type code (see Table 5-6)
\$11–\$12	Key pointer. If a subdirectory, the block number of the key block of the subdirectory. If a standard file, the block number of the index block of the file (or the data block if this is a seedling file).
\$13–\$14	Size of the file in blocks.
\$15–\$17	Size of the file in bytes (low-order bytes first).
\$18–\$19	The date on which this file was created (format: MMMDDDDD YYYYYYYM).
\$1A–\$1B	The minute (byte \$1A) and hour (byte \$1B) at which this file was created.
\$1C	The version number of ProDOS that created this file.
\$1D	The lowest version of ProDOS that is capable of using this file.
\$1E	Access code for this file (see Figure 5-1).
\$1F–\$20	For a binary file, the load address of the file; for a random-access text file, its record length.
\$21–\$22	The date on which this file was last modified (format: MMMDDDDD YYYYYYYM).
\$23–\$24	The minute (byte \$23) and hour (byte \$24) at which this file was created.
\$25–\$26	The block number of the key block of the directory that holds this file entry.

The only way to determine what type of file a particular file entry corresponds to is to examine the file type code that appears at relative position \$10 within the entry. Although 256 different codes are possible, only a few are commonly used by ProDOS, and it is these that are shown in Table 5-6. The three-character mnemonics used to represent these file types in a CATALOG listing are also shown in Table 5-6. For a list of all ProDOS file types, even the obscure ones, refer to the ProDOS Technical Reference Manual.

Table 5-6. ProDOS file type codes.

<i>File Type Code</i>	<i>CATALOG Mnemonic</i>	<i>Type of File</i>
\$04	TXT	ASCII text file (with bit 7 = 0)
\$06	BIN	Binary file
\$0F	DIR	Directory file
\$19	ADB	AppleWorks database file
\$1A	AWP	AppleWorks word processing file
\$1B	ASP	AppleWorks spreadsheet file
\$EF	PAS	Pascal file
\$F0	CMD	ProDOS added command file
\$FA	INT	Integer BASIC program file
\$FB	IVR	Integer BASIC variable file
\$FC	BAS	Applesoft program file
\$FD	VAR	Applesoft variable file
\$FE	REL	EDASM relocatable code file
\$FF	SYS	ProDOS system file

“Protecting” Files

ProDOS allow files to be protected using the LOCK command. If a file is locked, then it cannot be altered or renamed unless it is first unlocked. If a file is locked, then an asterisk will appear at the far left of the line in which the file name appears when the directory is catalogued.

ProDOS reserves a one-byte access code in its directory entries to indicate the write status of the file (at relative byte \$1E in each directory entry). Four bits in this byte are used to individually control the read, write, rename, and delete status of the file. A fifth bit acts as a flag to indicate whether the file has been modified since the last time it was backed up (it is the backup program’s responsibility to clear this bit to 0 when the file is backed up). These bits are described in Figure 5-1.

Unfortunately, there is no ProDOS command that can be used from Applesoft to adjust these bits individually. The LOCK command turns off the write, rename, and delete bits together and the UNLOCK command turns them all back on again. The bits can be changed, however, by directly reading the

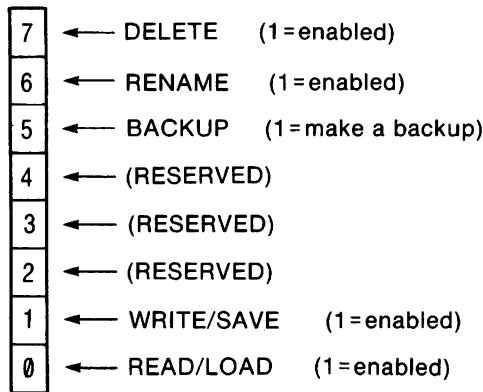


Figure 5-1. ProDOS access codes bit map.

block that contains the directory entry, changing the access code, and then writing the block back to diskette. The `READ.BLOCK` program listed in Table 5-9 will allow you to do this (this program will be described later on).

Storing File Data

The method ProDOS uses to keep track of where a standard file's data is stored on the diskette varies depending on the size of the file. ProDOS uses the following "woody" file classifications:

Seedling file	: 1 to 512 bytes (1 block or less)
Sapling file	: 513 to 131,072 (128K) bytes
Tree file	: 131,073 (128K + 1) to 16,777,215 (16M-1) bytes

ProDOS determines what type of file it is dealing with by examining the four highest bits of relative byte \$00 in the directory entry for the file: the number stored here is 1 for a seedling file, 2 for a sapling file, and 3 for a tree file.

ProDOS uses these three different file structures to reduce the amount of space needed to manage a file on the diskette to the absolute minimum. This permits ProDOS to deal with a file as quickly as possible and frees up valuable disk space for the storage of other files.

The directory entry's key pointer (relative bytes \$11 and \$12) points to the key block on the diskette for the file. Let's take a look at how ProDOS interprets this key block for each of the three types of files.

Seedling File. A seedling file, which, by definition, cannot exceed 512 bytes in length, obviously uses only one block on the diskette for data storage. It is this block that is pointed to by the key pointer. This means that the key block is, in fact, also the sole data block for the file.

Sapling File. The key pointer in the directory entry for this type of file points to an index block that contains an ordered list of the block numbers on the diskette that are used to store that file's data. Table 5-7 shows what an index block for a sapling file looks like. Since block numbers can exceed 255, two bytes are needed to store each block number. The low part of the block number is always stored in the first half of the block and the high part is stored 256 bytes further into the block. The maximum size of a sapling file is 128K; it cannot be larger than this since only 256 blocks (of 512 bytes each) can be pointed to by the index block.

Table 5-7. Map of the ProDOS index block for a sapling file.

<i>Byte Number</i>	<i>Meaning</i>
\$000	Block number of 0th data block (low)
\$001	Block number of 1st data block (low)
\$002	Block number of 2nd data block (low)
\$0FF	Block number of 255th data block (low)
\$100	Block number of 0th data block (high)
\$101	Block number of 1st data block (high)
\$102	Block number of 2nd data block (high)
\$1FF	Block number of 255th data block (high)

Tree File. If the file is a tree file, then the key pointer points to a master index block that contains an ordered list of the block numbers of up to 128 sapling-file-type index blocks. The structure of a master index block is shown in Table 5-8. Just as for sapling files, each of the index blocks pointed to by the master index block contains an ordered list of block numbers on the diskette that the file uses to store its data. The maximum size of a tree file is 16 megabytes (less one byte, which is reserved for an end-of-file marker)!

ProDOS takes care of all conversions that might become necessary if a file changes its type because it has either grown or shrunk. All this happens invisibly and it is not necessary to know what type of file is being dealt with unless special programs are being used that do not use the standard ProDOS commands to access files.

MLI—Accessing the Diskette Directly

ProDOS supports a special machine-language interface (MLI) protocol that makes it extremely simple for assembly-language programs to perform stan-

Table 5-8. Map of the ProDOS master index block for a tree file.

<i>Byte Number</i>	<i>Meaning</i>
\$000	Block number of 0th index block (low)
\$001	Block number of 1st index block (low)
\$002	Block number of 2nd index block (low)
\$07F	Block number of 127th index block (low)
\$100	Block number of 0th index block (high)
\$101	Block number of 1st index block (high)
\$102	Block number of 2nd index block (high)
\$17F	Block number of 127th data block (high)

dard diskette I/O commands. The MLI is explained in detail in Chapter 4 of Apple's ProDOS Technical Reference Manual.

The same general type of subroutine is used to invoke all MLI commands. The code used to invoke an MLI command looks like this (to review, "DFB" is a Merlin Pro assembler directive that causes the byte specified in the operand to be stored in memory):

```

JSR $BF00          ;Call the MLI
DFB CMDNUM         ; and execute this command #
DFB #<CMDLIST      ;Low part of address
DFB #>CMDLIST      ;High part of address
BCS ERROR          ;Error if carry flag set

```

where \$BF00 is the entry point to the MLI, CMDNUM is the command number that ProDOS has assigned to the requested command, and CMDLIST is the address of the parameter list associated with the command. (Recall from Chapter 2 that if you are using the Apple 6502 Assembler/Editor rather than Merlin Pro, then you should replace "#>" with "#<" and vice versa in the above example.) The parameter list contains the values of variables that the command needs in order to execute properly; result codes are also stored in the parameter list.

After the command is executed, control passes to the code that begins immediately after the three bytes stored after the "JSR \$BF00" instruction. If an error occurs, then the carry flag is set, the zero flag is cleared and the error code number is placed in the accumulator. You can transfer control to an error-handling subroutine by using a BCS instruction (as shown in the example) or a BNE instruction.

There are two MLI commands that can be used to read from and write to individual blocks on the diskette directly. The command numbers for these commands are \$80 (READ_BLOCK) and \$81 (WRITE_BLOCK). The parameter lists for these commands are identical and are constructed as follows:

- 1st byte: number of parameters (always \$03)
- 2nd byte: disk drive to be accessed
- 3rd byte: 512-byte data buffer address (low part)
- 4th byte: 512-byte data buffer address (high part)
- 5th byte: block number to be accessed (low part)
- 6th byte: block number to be accessed (high part)

The second byte in the parameter list contains information relating to the location of the diskette to be accessed. The number stored here is \$60 for the internal disk drive or \$E0 for the external disk drive.

For example, if a diskette is in the external disk drive and you want to read the contents of block number 260 on that diskette into a buffer beginning at location \$2000, you would use a program that looks like this:

```

JSR $BF00
DFB $80                      ;(Code for READ)
DFB #<CMDLIST
DFB #>CMDLIST
BCS IDERROR                  ;(Got it!)

.
RTS
IDERROR .                      ;(Didn't get it!)
.
RTS
CMDLIST DFB $03
        DFB $E0                ;External Drive
        DFB $00                ;Buffer address $2000
        DFB $20
        DFB $04                ;Block 260 ($0104)
        DFB $01
```

The same program can be used to write a block to the diskette simply by changing the command code from \$80 to \$81.

READ.BLOCK Program

Table 5-9 shows an extremely useful program called READ.BLOCK that can be used to examine any of the 280 blocks of data on a ProDOS diskette, to edit the contents of a block, and to write a modified block back to the diskette. READ.BLOCK makes use of the MLI READ_BLOCK and WRITE_BLOCK commands.

With READ.BLOCK, you can easily look at real examples of the types of blocks we have been discussing in this chapter, for example, the volume bit map, the directory blocks, the index blocks, and a file's data blocks themselves.

Table 5-9. READ.BLOCK—a program to read any block on a ProDOS diskette.

```

0  REM "READ.BLOCK"
100  FOR I = 768 TO 892: READ X:
      POKE I,X: NEXT
110  DEF FN MD(X) = X - 16 * INT
      (X / 16)
120  DEF FN M2(X) = X - 256 * INT
      (X / 256)
130  D$ = CHR$ (4)
140  BM = 279: REM NUMBER OF BLOC
      KS
150  TEXT : PRINT CHR$ (21): HOME
      : PRINT TAB( 16);: INVERSE
      : PRINT "READ BLOCK": NORMAL
      : PRINT TAB( 11);"(C) 1984
      GARY LITTLE"
160  VTAB 10: CALL - 958: PRINT
      "ENTER BASE BLOCK NUMBER (0-
      ";BM;: INPUT "): ";T$: IF T$
      = "" THEN 160
170  BL = INT ( VAL (T$)): IF BL
      = 0 AND T$ < > "0" THEN 16
      0
180  IF BL < 0 OR BL > BM THEN 1
      60
190  RW = 128
200  POKE 782, FN M2(BL): REM BL
      OCK# (LOW)
210  POKE 783, INT (BL / 256): REM
      BLOCK# (HIGH)
220  POKE 771,RW: REM READ=128 /
      WRITE=129
230  CALL 768
240  IF PEEK (8) < > 0 THEN PRINT
      : INVERSE : PRINT "DISK I/O
      ERROR": NORMAL : PRINT "PRES
      S ANY KEY TO CONTINUE: ";: GET
      A$: PRINT A$: GOTO 150
1000  VTAB 4: CALL - 958: PRINT
      TAB( 11);"CONTENTS OF BLOCK
      ";BL: PRINT : POKE 34,5
1010  Q = 1
1020  HOME : GOSUB 2000: CALL 79
      4:Q = Q + 1: IF Q = 5 THEN 1
      050
1030  IF PR = 0 THEN GET A$: IF
      A$ = CHR$ (27) THEN 1050

```

(continued)

Table 5-9. READ.BLOCK—a program to read any block on a ProDOS diskette (continued).

```

1040 GOTO 1020
1050 Q = Q - 1: PR = 0: PRINT D$;
      "PR#0": B = 0
1060 HTAB 1: VTAB 23: CALL - 9
      58: PRINT "ENTER COMMAND (B,
      C,D,E,N,P,Q,W,HELP): ";: GET
      A$: IF A$ = CHR$ (13) THEN
      A$ = " "
1070 PRINT A$
1080 IF A$ < > "D" THEN 1110
1090 Q = Q - 1: IF Q = 0 THEN Q =
      4
1100 HOME : GOSUB 2000: CALL 79
      4: GOTO 1060
1110 IF A$ = "H" THEN 5000
1120 IF A$ = "Q" THEN 1260
1130 IF A$ = "E" THEN 1270
1140 IF A$ = "P" THEN 1220
1150 IF A$ = "N" THEN 1240
1160 IF A$ = "B" THEN 150
1170 IF A$ = "C" THEN VTAB 23:
      CALL - 958: PRINT TAB( 6)
      ;; INVERSE : PRINT "TURN ON
      PRINTER IN SLOT #1": NORMAL
      : PR = 1: PRINT D$; "PR#1": PRINT
      : GOTO 1000
1180 IF A$ < > "W" THEN 1210
1190 POKE 782,BL: POKE 771,129:
      VTAB 23: CALL - 958: PRINT
      "PRESS 'Y' TO VERIFY WRITE:
      ";: GET A$: IF A$ = CHR$ (1
      3) THEN A$ = " "
1200 PRINT A$: IF A$ = "Y" THEN
      CALL 768: RW = 128: VTAB 23:
      CALL - 958: PRINT "WRITE C
      OMPLETED. PRESS ANY KEY: ";:
      GET A$: GOTO 1060
1210 GOTO 5000
1220 BL = BL - 1: IF BL = - 1 THEN
      BL = 279
1230 GOTO 190
1240 BL = BL + 1: IF BL > 279 THEN
      BL = 0
1250 GOTO 190
1260 TEXT : HOME END

```

(continued)

Table 5-9. READ.BLOCK—a program to read any block on a ProDOS diskette (continued).

```

1270 V = 8:H = 3: VTAB 5: PRINT
      TAB( 6);: INVERSE : PRINT "
      I=UP M=DOWN J=LEFT K=RIGHT":
      NORMAL
1280 HTAB 1: VTAB 23: CALL - 9
      58: PRINT TAB( 6);"PRESS ";
      : INVERSE : PRINT "ESC";: NORMAL
      : PRINT " TO LEAVE EDITOR"
1290 REM
1300 GOSUB 1500: GET A$
1310 LC = 16384 + 128 * (Q - 1) +
      8 * V + H:Y = PEEK (LC):X =
      ASC (A$)
1320 IF A$ = CHR$ (27) THEN HTAB
      1: VTAB 5: CALL - 868: GOTO
      1060
1330 IF A$ < > "I" THEN 1370
1340 B = 0:V = V - 1: IF V > =
      0 THEN 1300
1350 V = 15:Q = Q - 1: IF Q < 1 THEN
      Q = 4
1360 GOSUB 2000: HOME CALL 79
      4: GOTO 1280
1370 IF A$ = "J" THEN B = 0:H =
      H - 1: IF H = - 1 THEN H =
      7
1380 IF A$ = "K" THEN B = 0:H =
      H + 1: IF H = 8 THEN H = 0
1390 IF A$ < > "M" THEN 1430
1400 B = 0:V = V + 1: IF V < 16 THEN
      1300
1410 V = 0:Q = Q + 1: IF Q = 5 THEN
      Q = 1
1420 GOTO 1360
1430 IF B = 0 THEN Y = FN MD(Y
      ) + 16 * (X - 48) * (X < =
      57) + 16 * (X - 55) * (X > =
      65)
1440 IF B = 1 THEN Y = 16 * INT
      (Y / 16) + (X - 48) * (X < =
      57) + (X - 55) * (X > = 65)

1450 X = ASC (A$): IF (X > = 4
      8 AND X < = 57) OR (X > =
      65 AND X < = 70) THEN PRINT
      A$;: POKE ( PEEK (40) + 256 *

```

(continued)

Table 5-9. READ.BLOCK—a program to read any block on a ProDOS diskette (continued).

```

      PEEK (41) + 31 + H),Y: POKE
      LC,Y: IF B = 0 THEN CALL 64
      500:B = 1
1460  IF X = 8 AND B = 1 THEN B =
      0
1470  IF X = 21 AND B = 0 THEN B
      = 1
1480  GOTO 1300
1490  CALL - 167
1500  VTAB V + 6: HTAB 3 * H + 7
      + B: RETURN
2000  IF Q = 1 THEN POKE 795,0:
      POKE 799,64
2010  IF Q = 2 THEN POKE 795,12
      8: POKE 799,64
2020  IF Q = 3 THEN POKE 795,0:
      POKE 799,65
2030  IF Q = 4 THEN POKE 795,12
      8: POKE 799,65
2040  RETURN
5000  HOME : PRINT TAB( 10);"SU
      MMARY OF COMMANDS": PRINT TAB(
      10);"=====": PRINT
5010  PRINT "B--RESET BASE BLO
      CK"
5020  PRINT "C--COPY BLOCK CON
      TENTS TO PRINTER"
5030  PRINT "D--DISPLAY PREVIO
      US 1/4 BLOCK"
5040  PRINT "E--EDIT THE CURRE
      NT BLOCK"
5050  PRINT "N--READ THE NEXT
      BLOCK"
5060  PRINT "P--READ THE PREVI
      OUS BLOCK"
5070  PRINT "Q--QUIT THE PROGR
      AM"
5080  PRINT "W--WRITE THE BLOC
      K TO DISK"
5090  PRINT : PRINT "PRESS ANY K
      EY TO CONTINUE: ";: GET A$: PRINT
      A$: GOTO 1100
8000  DATA 32,0,191,128,10,3,144
      ,8,176,11,3,96,0,64,0,0,169,
      0,133,8,96,169,1,133,8,96,16
      9,0,133,6

```

(continued)

Table 5-9. READ.BLOCK—a program to read any block on a ProDOS diskette (continued).

```

8010 DATA 169,64,133,7,162,0,16
      0,0,56,165,7,233,64,32,218,2
      53,165,6,32,218,253,169,186,
      32,237,253,169,160,32,237
8020 DATA 253,177,6,32,218,253,
      169,160,32,237,253,200,192,8
      ,208,241,169,160,32,237,253,
      160,0,177,6,9,18,201,160,17
      6
8030 DATA 2,169,174,32,237,253,
      200,192,8,208,238,169,141,32
      ,237,253,24,165,6,105,8,133,
      6,165,7,105,0,133,7,232
8040 DATA 224,16,208,168,96

```

You should be careful when writing a block to a diskette, however, as it is easy to accidentally render the diskette unreadable; you should always experiment on a backup copy of the original diskette.

When READ.BLOCK is first run, you will be asked to enter a base block number. After this information has been provided, the block corresponding to that location on the diskette will be read into memory and displayed on the screen in a special format. Because of 40-column screen size limitations, only one-quarter of the block can be represented at once (you have to press the “D” key to display the other quarters).

The contents of a block are displayed in 64 rows, each of which contains an offset address from the beginning of a block followed by the hexadecimal representations of the eight bytes stored from that location onward in the block. At the far right of each row are the ASCII representations of each of these eight bytes. Note that only 16 rows are displayed at any one time.

After the entire block has been displayed, you will be asked to enter one of eight commands. The meanings of each of these commands are as follows:

- “B”—reset the base block
- “C”—copy the contents of the block to the printer (in port 1)
- “D”—display the next quarter of the current block
- “E”—edit the current block
- “N”—read and display the next block on the diskette
- “P”—read and display the previous block on the diskette
- “Q”—quit the program
- “W”—write the block back onto the diskette

The functions that most of these commands perform are obvious. The only “tricky” one is the “E” (Edit) command. When the Edit command is entered,

the cursor will move into the middle of the 8x16 array of hexadecimal digits that represent the contents of one-quarter of the block. To change any of these digits, use the I, J, K, and M keys to move the cursor up, left, right, and down, respectively, and then enter the new two-digit hexadecimal entry for that position. You can leave editing mode at any time by pressing the ESC key. Once you have left editing mode, you can save the changes to diskette by using the "W" (Write) command.

Further Reading for Chapter 5

Standard reference works . . .

ProDOS Technical Reference Manual, Apple Computer, Inc., 1983.

ProDOS User's Manual, Apple Computer, Inc., 1983.

BASIC Programming with ProDOS, Apple Computer, Inc., 1983.

On Applesoft and ProDOS . . .

R.D. Norling, "Running APPLESOFT Programs with ProDOS", *Call - A.P.P.L.E.*, July 1984, pp. 43-50.

On the internal structure of ProDOS. . .

B. Sander-Cederlof, "Commented Listing of ProDOS", *Apple Assembly Line*:

a. \$F90C-\$F995, \$FD00-\$FE9A, \$FEBE-\$FFFF : December 1983, pp. 2-11

b. \$F800-\$F90B, \$F996-\$FEBD : November 1983, pp. 2-14

Reviews of ProDOS . . .

R. Moore, "ProDOS", *Byte*, February 1984, pp. 252-262.

C. Fretwell, "ProDOS and Friends", *Call - A.P.P.L.E.*, January 1984, pp. 65-69.

G.W. Charpentier and D. Sparks, "Taking the 'Pro' out of ProDOS", *Call - A.P.P.L.E.*, November 1983, pp. 9-14.

S. Mossberg, "An Introduction to ProDOS", *Nibble*, March 1984, pp. 153-159.

T. Weishaar, "Breaking the Floppy Barrier: An Introduction to Apple's ProDOS", *Softalk*, January 1984, pp. 112-118.

6

Character Input and the Keyboard

The //c, like most other microcomputers, usually deals with information that is delivered to it in one-byte (8-bit) chunks (from a keyboard or a disk drive, for example). This information is commonly referred to as “character” input because the bytes usually represent the encoded representations of letters of the alphabet, numbers, and other printable characters. Although any encoding scheme that the input device cares to use could be dealt with by the //c, it is the American National Standard Code for Information Interchange (ASCII) standard that is usually used to encode characters. Two other incompatible encoding schemes, Extended Binary-Coded Decimal Interchange Code (EBCDIC) and Baudot, are also in widespread use, the first by all large IBM computers (except the IBM PC series) and compatibles and the second by some older TeleType terminals.

ASCII is a seven-bit code and is used by virtually all microcomputers. A total of 128 (2^7) codes are defined by the ASCII standard and each code fits nicely into one byte with one bit (bit 7) to spare. Table 6-1 contains a list of these codes, their standard names or symbols, and the keys on the keyboard (or combination of keys) that must be pressed to enter them.

When the //c performs character input/output operations, the ASCII code for the character is stored in bits 0 through 6 of the byte being inputted or outputted and bit 7 of the byte is normally set equal to “1”. Since a “1” in bit 7 is often used to indicate that the value stored in that byte is negative, this “variant” of ASCII is called “negative ASCII”; if bit 7 is 0, then “positive ASCII” is being used.

Note that all but the first 32 ASCII codes and ASCII code 127 (rubout) are used to represent visible symbols. The first 32 codes are called “control characters” and are usually sent to a video display or printer controller to cause it to perform some special action. Some of the more important control characters on the //c are as follows (in negative ASCII):

Table 6-1. American National Standard Code for Information Interchange (ASCII) character codes.**ASCII**

Code				
Hex	Dec	Symbol		Keys to Press
\$00	000	NUL	(Null)	CONTROL @
\$01	001	SOH	(Start of header)	CONTROL A
\$02	002	STX	(Start of text)	CONTROL B
\$03	003	ETX	(End of text)	CONTROL C
\$04	004	EOT	(End of transmission)	CONTROL D
\$05	005	ENQ	(Enquiry)	CONTROL E
\$06	006	ACK	(Acknowledge)	CONTROL F
\$07	007	BEL	(Bell)	CONTROL G
\$08	008	BS	(Backspace)	LEFT-ARROW or CONTROL H
\$09	009	HT	(Horizontal tabulation)	TAB or CONTROL I
\$0A	010	LF	(Line feed)	DOWN-ARROW or CONTROL J
\$0B	011	VT	(Vertical tabulation)	UP-ARROW or CONTROL K
\$0C	012	FF	(Form feed)	CONTROL L
\$0D	013	CR	(Carriage return)	RETURN or CONTROL M
\$0E	014	SO	(Shift out)	CONTROL N
\$0F	015	SI	(Shift in)	CONTROL O
\$10	016	DLE	(Data link escape)	CONTROL P
\$11	017	DC1	(Device control 1)	CONTROL Q
\$12	018	DC2	(Device control 2)	CONTROL R
\$13	019	DC3	(Device control 3)	CONTROL S
\$14	020	DC4	(Device control 4)	CONTROL T
\$15	021	NAK	(Negative acknowledge)	RIGHT-ARROW or CONTROL U
\$16	022	SYN	(Synchronous idle)	CONTROL V
\$17	023	ETB	(End of transmission block)	CONTROL W
\$18	024	CAN	(Cancel)	CONTROL X
\$19	025	EM	(End of medium)	CONTROL Y
\$1A	026	SUB	(Substitute)	CONTROL Z
\$1B	027	ESC	(Escape)	ESC or CONTROL [
\$1C	028	FS	(Field separator)	CONTROL \
\$1D	029	GS	(Group separator)	CONTROL]
\$1E	030	RS	(Record separator)	CONTROL ^
\$1F	031	US	(Unit separator)	CONTROL _
\$20	032		(Space)	SPACE BAR
\$21	033	!		SHIFT 1
\$22	034	"		SHIFT '
\$23	035	#		SHIFT 3
\$24	036	\$		SHIFT 4
\$25	037	%		SHIFT 5
\$26	038	&		SHIFT 7

Table 6-1. American National Standard Code for Information Interchange (ASCII) character codes (continued).**ASCII**

Code			
Hex	Dec	Symbol	Keys to Press
\$27	039	'	'
\$28	040	(SHIFT 9
\$29	041)	SHIFT 0
\$2A	042	*	SHIFT 8
\$2B	043	+	SHIFT =
\$2C	044		
\$2D	045		
\$2E	046	.	.
\$2F	047	/	/
\$30	048	0	0
\$31	049	1	1
\$32	050	2	2
\$33	051	3	3
\$34	052	4	4
\$35	053	5	5
\$36	054	6	6
\$37	055	7	7
\$38	056	8	8
\$39	057	9	9
\$3A	058		SHIFT ;
\$3B	059	;	;
\$3C	060	<	SHIFT ,
\$3D	061	=	=
\$3E	062	>	SHIFT .
\$3F	063	?	SHIFT /
\$40	064	@	SHIFT 2
\$41	065	A	SHIFT A
\$42	066	B	SHIFT B
\$43	067	C	SHIFT C
\$44	068	D	SHIFT D
\$45	069	E	SHIFT E
\$46	070	F	SHIFT F
\$47	071	G	SHIFT G
\$48	072	H	SHIFT H
\$49	073	I	SHIFT I
\$4A	074	J	SHIFT J
\$4B	075	K	SHIFT K
\$4C	076	L	SHIFT L
\$4D	077	M	SHIFT M
\$4E	078	N	SHIFT N
\$4F	079	O	SHIFT O
\$50	080	P	SHIFT P
\$51	081	Q	SHIFT Q

Table 6-1. American National Standard Code for Information Interchange (ASCII) character codes (continued).

ASCII Code			Keys to Press
Hex	Dec	Symbol	
\$52	082	R	SHIFT R
\$53	083	S	SHIFT S
\$54	084	T	SHIFT T
\$55	085	U	SHIFT U
\$56	086	V	SHIFT V
\$57	087	W	SHIFT W
\$58	088	X	SHIFT X
\$59	089	Y	SHIFT Y
\$5A	090	Z	SHIFT Z
\$5B	091	[[
\$5C	092	\	\
\$5D	093]]
\$5E	094	^	SHIFT 6
\$5F	095		SHIFT -
\$60	096		'
\$61	097	a	A
\$62	098	b	B
\$63	099	c	C
\$64	100	d	D
\$65	101	e	E
\$66	102	f	F
\$67	103	g	G
\$68	104	h	H
\$69	105	i	I
\$6A	106	j	J
\$6B	107	k	K
\$6C	108	l	L
\$6D	109	m	M
\$6E	110	n	N
\$6F	111	o	O
\$70	112	p	P
\$71	113	q	Q
\$72	114	r	R
\$73	115	s	S
\$74	116	t	T
\$75	117	u	U
\$76	118	v	V
\$77	119	w	W
\$78	120	x	X
\$79	121	y	Y
\$7A	122	z	Z
\$7B	123	{	SHIFT [
\$7C	124		SHIFT \

Table 6-1. American National Standard Code for Information Interchange (ASCII) character codes (continued).

ASCII			
Code			
Hex	Dec	Symbol	Keys to Press
\$7D	125	}	SHIFT]
\$7E	126	~	SHIFT '
\$7F	127	■ (Rubout)	DELETE

\$87 (bell)—causes the speaker to beep

\$88 (backspace)—causes the cursor to move back one position

\$8A (line feed)—causes the cursor to move down one line

\$8D (carriage return)—causes the cursor to move to the beginning of the current line

Some of the names associated with the control characters (see Table 6-1) are somewhat archaic in that they refer to various aspects of the operation of old TeleType terminals. Other names relate to the codes used by certain standard data-interchange protocols that are used in telecommunications networks (for example, Start of Text (STX), End of Text (ETX), and Cancel (CAN)).

In this chapter, we will take a look at how the //c requests and reads character input from any device interfaced to it, including the keyboard. In doing so, we will examine the built-in ROM subroutines that the //c normally uses whenever it requires character input.

You will be able to follow this chapter a lot more easily if you have by your side a copy of the source code listings for the Apple //c monitor ROM. They can be found in "The Apple //c Reference Manual", volume 2.

Standard Character Input Subroutines

There are three special, general-purpose character input subroutines in the //c's system monitor that are used to fetch characters so that they can be used and interpreted by other parts of the system, including Applesoft and the system monitor. These routines are usually referred to by the symbolic names of RDKEY, RDCHAR, and GETLN. (A fourth subroutine, ESCRDKEY, is functionally identical to RDCHAR.) They, in turn, usually make use of two other subroutines that are used to read information from the keyboard; these are called KEYIN and C3KEYIN. Each of these subroutines are briefly described in Table 6-2. Let's take a closer look at them.

Table 6-2. Built-in input subroutines.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$FD0C	(64780)	RDKEY	Reads a character from the currently active input device and places its negative ASCII code in the accumulator.
\$FD35	(64821)	RDCHAR	Both use RDKEY to read a character from the currently active input device and both enable escape sequences.
\$CCED	(52461)	ESCRDKEY	
\$FD1B	(64795)	KEYIN	Keyboard input routine used when 80-column firmware is not being used. The negative ASCII code for the character is returned in the accumulator.
\$C305	(51446)	C3KEYIN	Keyboard input routine used when 80-column firmware is being used. The negative ASCII code for the character is returned in the accumulator.
\$FD6A	(64874)	GETLN	Reads a line of information into the input buffer at \$200 by making repeated calls to ESCRDKEY.

Reading One Character

RDKEY (\$FD0C)

Because RDKEY is the fundamental character input subroutine that is eventually called by the other two, it is the most important of the three. This subroutine is used to scan any input device that has been designated as being active (usually, but not necessarily, the keyboard) until a character has been entered, and to return the ASCII code for that character (with its high bit set to one) in the 65C02's accumulator. The Applesoft GET command calls RDKEY directly.

As soon as RDKEY is called, it loads the character stored at the currently active video position (as calculated from the values of CH (\$24) and CV (\$25), the horizontal and vertical cursor coordinates). The code that does this looks like this:

```
LDY CH           ;Get horizontal position
LDA (BASL),Y     ;Get the screen byte
NOP
                 ;(Seven more NOPs)
```


where BASL (\$28) is the first of two zero page locations that together contain the base address for the line number held in CV. (See Chapter 7.)

After the screen character has been obtained, the following code is executed:

```
JMP (KSWL)
```

which effectively passes control to the body of a user-selectable input subroutine whose address is held at KSWL (\$38) and KSWH (\$39). This input subroutine is responsible for returning the ASCII code for an inputted character as soon as the input device being used makes one available. For the purposes of this discussion, we will assume that the input device is the //c's keyboard. We will see later on how other input devices can be linked into the RDKEY subroutine instead by simply storing the address of the input subroutine for the alternative input device at KSWL and KSWH.

The //c's ProDOS disk operating system is integrated into the system by storing the address of its special input subroutine at KSWL and KSWH. This input subroutine will read input from either a diskette file or the keyboard, depending on whether or not a ProDOS READ command is in effect. It will also cause special disk operations to be performed if a valid ProDOS command is entered from the keyboard (for example, LOAD a file and CATALOG the diskette). When it reads information from the keyboard, it uses one of the //c's two built-in subroutines available for this purpose.

The keyboard input subroutine that is used will depend on whether the //c's internal 80-column firmware ROM is being used. This firmware is not on when you first turn on the //c but can be selected by entering a PR#3 command from Applesoft. Once you have selected the 80-column firmware in this way, you can flip between an 80-column display and a 40-column display by using the two-keystroke "escape sequence"

```
ESC 4
```

to go from 80-column mode to 40-column mode and

```
ESC 8
```

to go from 40-column mode to 80-column mode. (An escape sequence is entered by pressing the ESC key, releasing it, and then pressing the second key; the [return] key must not be pressed.)

You can usually tell whether the 80-column firmware is being used by looking at the cursor. If it's a blinking "checkerboard," then the 80-column firmware is not in use; if it's a nonflashing, inverse-video square, then it is. The 80-column firmware can be deactivated by entering an ESC [control-Q] sequence from the keyboard or printing a [control-U] character; either method returns you to standard 40-column mode. (You can also use the PR#0 command to return to standard 40-column mode, but only if ProDOS is not being used. This method will not work at all on the Apple //e, however.)

We will be looking at the video display modes in considerably more detail in Chapter 7.

Keyboard Input

If the 80-column firmware on the //c is not being used, then the //c usually uses a subroutine called KEYIN (\$FD1B) to handle keyboard input. If the 80-column firmware is active, then the C3KEYIN (\$C305) subroutine is used instead. C3KEYIN, however, simply calls the KEYIN subroutine to handle keyboard input. KEYIN behaves slightly differently for each video mode, however; it determines which mode is active by examining the contents of certain memory locations that are set up when the video modes are initialized.

The first thing that KEYIN does is to set up a cursor on the screen by calling SHOWCUR (\$CC4C). SHOWCUR examines location CURSOR (\$7FB) to determine what type of cursor to set up. If the number stored there is \$FF, then a “checkerboard” cursor will be displayed; this is the cursor that is normally used when the standard 40-column display is active. If the number stored at CURSOR is 0, then the cursor used is an inverse block; this is the one that is normally used when the 80-column firmware is active. If you store any other number at CURSOR, you can generate all sorts of interesting blinking cursors. For example, by storing 186 at CURSOR, the cursor will be a blinking colon.

The blinking cursors used by the //c are generated by using software timing loops. For example, to display the checkerboard cursor, the //c alternates between the display of the checkerboard character (ASCII code \$FF) and the true screen character at fixed intervals.

When a key that generates an ASCII code is entered, the cursor is removed by placing the screen character back on the screen, and then the ASCII code representing the entered key is placed in the 65C02’s accumulator (with the high bit set to one).

At this point, KEYIN moves on to the GOTKEY (\$FD25) subroutine. This subroutine examines bit 3 of location VMODE (\$4FB), the escape enable bit, to determine whether it should just pass the keyboard character along to the caller or whether to process it further.

If bit 3 of VMODE is 0, then KEYIN ends with the character code still in the accumulator. If it is 1, however, then GOTKEY checks to see if the character that was entered was the ESC key. If it wasn’t, then the escape enable bit of VMODE is turned off and KEYIN ends.

Escape Sequences

If, however, the ESC key is pressed, then KEYIN does something quite different: control passes to NEWESC (\$CCCC), which causes the cursor to change to an inverse “+” sign and “escape mode” to be turned on. Whenever the //c is in this mode, it reads the keyboard once again and then executes a special function dictated by the key that is read. This two-key combination is commonly referred to as an “escape sequence.” A list of all of the valid escape sequences and the functions they perform are listed in Table 6-3.

Table 6-3. Escape sequences.

<i>Escape Sequence</i>	<i>Description</i>
ESC @	Clears the video screen window and places the cursor in the top left-hand corner.
ESC A	Moves the cursor one position to the right.
ESC B	Moves the cursor one position to the left.
ESC C	Moves the cursor down one line.
ESC D	Moves the cursor up one line (if not already at top).
ESC E	Clears the screen from the current cursor position to the end of the line. The cursor position does not change.
ESC F	Clears the screen from the current cursor position to the end of the window. The cursor position does not change.
ESC I ESC ↑	Moves the cursor up one line and keeps escape mode active.
ESC J ESC ←	Moves the cursor one position to the left and keeps escape mode active.
ESC K ESC →	Moves the cursor one position to the right and keeps escape mode active.
ESC M ESC ↓	Moves the cursor down one line and keeps escape mode active.
ESC 4	Switches to 40-column mode from 80-column mode.
ESC 8	Switches to 80-column mode from 40-column mode.
ESC [control-Q]	Deselects the 80-column firmware and returns to standard 40-column mode.
ESC [control-D]	Disables the printing of control characters when the 80-column firmware is active (other than carriage return, line feed, backspace, and bell).
ESC [control-E]	Enables the printing of all control characters.

Most of the escape sequences that have been defined on the //c are used to move the cursor around the screen or to affect the video display in some way and are adequately explained in Table 6-3. Two of them are somewhat unusual: they are ESC [control-D] and ESC [control-E].

ESC [control-D] is used to disable any special interpretation of control characters that are sent to the standard output subroutine other than the

codes for carriage return (\$8D), line feed (\$8A), backspace (\$88), and bell (\$87). As we will see in Chapter 7, the //c reacts in special ways to several other control characters that are sent to its output subroutine when the 80-column firmware is being used. For example, if a [control-U] character is printed, you can move from the 80-column display to a 40-column display. This is a handy way of exiting 80-column mode, but what if you happen to be communicating with a remote computer through the modem port when you receive a [control-U] character that the computer uses to indicate that a file is ready to be transferred? Before you know it, you will pop out of 80-column mode and you will be left scratching your head wondering what happened. To avoid this type of trouble it is best to enter the ESC [control-D] sequence before running such a program. You can re-enable the handling of the control codes later by entering the ESC [control-E] sequence.

In general, escape mode ends immediately after the key after ESC has been pressed and, if you want to re-enter escape mode, you must press ESC once again. The I,J,K,M and arrow-key sequences, however, behave somewhat differently. If you enter any of these sequences, then escape mode remains active until any other key that generates an ASCII code that is not part of an escape sequence is pressed. This means that you can quickly move the cursor around the screen by pressing ESC once and then pressing any sequence of cursor-movement keys until the cursor is properly positioned. You can then press another key (the space bar is convenient) to leave escape mode.

When escape mode ends, the keyboard character that was last pressed is returned in the accumulator and KEYIN ends.

RDCHAR (\$FD35) and ESCRDKEY (\$CCED)

The RDCHAR and ESCRDKEY subroutines simply turn on the escape enable bit in VMODE (\$4FB) before jumping to the start of the RDKEY subroutine to get input. This tells the RDKEY subroutine to handle any escape sequences that may be entered before a standard character code is returned.

The RDCHAR subroutine is actually identical to the ESCRDKEY subroutine since it simply jumps to the start of ESCRDKEY as soon as it is called. It has been made available for the sake of compatibility with the earlier Apple II models which had a similar subroutine located at \$FD35.

Reading a Line of Characters

RDKEY, RDCHAR, and ESCRDKEY read only one character at a time. A much more useful and general subroutine is one that allows you to enter a whole line of information at once (a line being defined as a series of characters that is entered before [return] is pressed). Such a subroutine does exist on the //c and is called GETLN (\$FD6A).

The GETLN subroutine is used by the //c whenever you are entering commands in the system monitor or in Applesoft direct mode. In addition, the Applesoft INPUT command uses this subroutine directly.

As soon as GETLN is called, a special symbol, called a prompt symbol, is displayed. The code for this symbol is always read from PROMPT (\$33). This symbol serves two purposes: it tells you what part of the //c is currently active (the system monitor or Applesoft, for example) and it reminds you that the //c is expecting you to enter a line of information. Table 6-4 sets out the various prompts symbols commonly used by the //c.

Table 6-4. //c prompt symbols.

<i>Prompt Symbol</i>	<i>Meaning</i>
*	the system monitor is waiting for a command.
]	Applesoft is waiting for you to enter a command or a program line.
?	Applesoft is waiting for you to respond to an INPUT statement.

Note: The ASCII code for the prompt symbol is kept in PROMPT (\$33).

After the prompt symbol has been displayed, GETLN calls ESCRDKEY again and again until the [return] key is pressed. The characters returned by the series of RDCHAR calls are stored in consecutive locations in a 256-byte character input buffer located in page two of memory beginning at IN (\$200). When [return] is pressed, the subroutine ends and the number of characters in the buffer is returned in the X register.

When a line is entered using GETLN, all those escape sequences that are normally available can be used. In addition, GETLN supports several simple editing commands that can be used when the line is being entered. These editing commands will now be discussed.

Left-Arrow Key. This key allows you to backspace over the previous item in the input buffer and, thus, to remove it from the buffer. The cursor moves one position to the left on the video screen when the left-arrow key is pressed.

Right-Arrow Key. This key allows you to copy the character on the video screen beneath the cursor into the input buffer.

Ctrl-X. This key allows you to erase everything that is currently in the input buffer. When it is pressed, a backslash ("\") will be displayed after the characters that have already been typed in and the cursor will be placed at the far left of the next line on the screen. Note that the line will automatically

be canceled like this if you attempt to enter more than 255 characters before pressing [return]. Beeps will be sounded after every character entered after the 248th one to remind you that the buffer is almost full.

Return. This key indicates to GETLN that the current line is completed and is to be entered.

Changing Input Devices: The Input Link

The most common source of character input to the //c is the keyboard. It is possible, however, to redirect it to other input devices which can be connected to the //c through its built-in expansion ports. A familiar example of such a source is the //c's disk drive.

The //c uses a very flexible method for handling the problems associated with having many possible sources of character input. Even though the source of the input may vary, calls are still always made to the RDKEY subroutine whenever a character from any device, in general, is required. To activate a particular device, the destination of a jump instruction that RDKEY uses to locate the character input subroutine is set to the address of the device's input subroutine. This means that your program's input commands (for example, INPUT and GET in Applesoft) can always be used regardless of the source of input.

Let's take a closer look at the mechanics of this procedure. We saw earlier that whenever RDKEY is called to obtain another character, control ultimately passes to an instruction that looks like this:

```
JMP ($0038)
```

The addressing mode used by the jump instruction here is called "indirect." This means that the destination of the jump is not location \$38 itself but rather the address stored at locations \$38 (low byte) and \$39 (high byte). This address is normally a subroutine within ProDOS that ultimately calls KEYIN (\$FD1B) or C3KEYIN (\$C305), the system monitor's standard keyboard input routines (unless input is being requested from a diskette file). By simply changing the address stored at \$38/\$39, however, you can force the //c to execute any subroutine that you want whenever input is requested, including one associated with an alternative input device.

The symbolic name for locations \$38/\$39 is KSW (for keyboard switch); \$38 by itself is called KSWL and \$39 is called KSWH. Since these locations are used to incorporate new input routines into the system, KSW is commonly referred to as the "input link" or "input hook."

The address of the input subroutine for a peripheral input device is usually placed in KSWL and KSWH by using the Applesoft "IN#s" command. This command causes the //c to transfer control to a program beginning at location \$Cs00 (where "s" is a valid port number) that is the first location in a ROM

area reserved for that port. The program in the ROM area will modify KSW so that it will point to a new input routine also contained in that ROM. Note that if an IN#0 command is entered, then the address of KEYIN (\$FD1B), the //c's standard 40-column input subroutine, will be stored at KSWL and KSWH.

You can also change the input hook by using the Applesoft POKE command to store the address of the new input routine directly into KSW at \$38 and \$39; this address can be in a ROM area or a RAM area.

How About Output?

You may well be wondering whether the //c uses the same method to handle its output that it uses to handle its input. The answer is, you guessed it, "yes," but we're going to defer discussion of output until Chapter 7. For those of you who just can't wait, the //c uses an output link called CSW (\$36/\$37) to point to the output subroutine that is to take control whenever the standard output subroutine, COUT (\$FDED), is called. The PR# command can be used to transfer control to a port in much the same way that IN# can be.

Designing a KSW Input Subroutine

A KSW input subroutine must be designed carefully to ensure that it adheres to certain rules that restrict its usage of 65C02 registers. The most important rule is that when the subroutine ends, the inputted character must be contained in the accumulator with its high-order bit set to one. Furthermore, the X and Y registers must contain the same values they held when the subroutine was first entered. Thus, if X and Y are to be changed by the KSW subroutine, they must first be saved in a safe place (such as the stack) and then restored just before the subroutine ends.

Replacing the Keyboard Input Subroutine

As we saw earlier, the //c comes with a built-in keyboard input subroutine called KEYIN (\$FD1B). This subroutine takes care of setting up the cursor and of scanning the keyboard until a key has been pressed. There is nothing magic about this particular subroutine, however, and you could easily replace it with another program that would still get input from the keyboard, but would do it differently. In fact, this is essentially what is done whenever you enter a PR#3 command to turn on the //c's 80-column display. As we have seen, when this is done, RDKEY uses a new keyboard input routine called C3KEYIN which changes the type of cursor that is used.

You can use your own imagination to dream up some useful features that could be added to a keyboard input subroutine. Some interesting ones to think about are as follows:

- The ability to prevent certain characters from being entered.
- Allowing additional escape sequences.
- Allowing for macro keys. (A macro key is one that, when pressed, causes a whole string of characters to be entered.)

Later in this chapter, after we have seen how to read the keyboard, we will present some examples of modifying the keyboard input subroutine to meet special requirements such as these.

It is fairly simple to redefine the keyboard input subroutine so that it operates properly in whatever video mode is currently active. In fact, only five basic steps need be performed:

- Set up a cursor.
- Wait for a key to be pressed (while blinking the cursor, if desired).
- Remove the cursor.
- Handle escape sequences (if desired).
- Return with the key code in the accumulator.

The //c's KEYIN (\$FD1B) subroutine is, of course, the perfect example of a KSW subroutine that performs these five steps when it is called. We looked at how KEYIN works earlier in this chapter, but let's quickly relate its activities to each of the five steps.

KEYIN first sets up a cursor (step 1) by calling SHOWCUR (\$CC4C) to display the character stored at CURSOR (\$7FB) at the currently active cursor position. It then repeatedly calls the UPDATE (\$CC70) subroutine until a key has been pressed (step 2). The UPDATE subroutine takes care of blinking the cursor at the proper rate (if necessary) and will remove the cursor after a key is pressed by calling STORY (\$C3B3) with the original screen character in the accumulator (step 3). After a key has been pressed, KEYIN calls the GOTKEY (\$FD25) subroutine to handle any valid escape sequences that may be entered (step 4). Finally, KEYIN ends with the ASCII code for the keyboard character in the accumulator with its high bit set to 1 (step 5).

The basic subroutines used by KEYIN to perform its duties are summarized in Table 6-5. These are built-in to the //c's ROM and may be used by your own programs, if desired.

ProDOS and the Input Link

The ability to change the KSW input link is somewhat restricted if ProDOS is active. (Similar restrictions apply if the CSW output link is to be changed.) When ProDOS is first activated, the address stored in the KSW input link is placed in another input link located within ProDOS itself. A special KSW input subroutine is then installed that is responsible for detecting and executing any ProDOS commands that are entered (when Applesoft direct mode

Table 6-5. Built-In subroutines used by KEYIN (\$FD1B).

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$CC4C	(52300)	SHOWCUR	Displays the character stored at CURSOR (\$7FB) at the current cursor position and designates it as the active cursor.
\$CC70	(52336)	UPDATE	Causes the cursor to blink at the proper rate (if non-zero) and scans the keyboard for input. If a key is pressed, then the N-flag in the processor status register is set to 1, the character code is placed in the accumulator, and the cursor is removed by calling STORY (\$C3B3) to restore the original screen character. If a key is not pressed, then the N-flag is set to 0.
\$C3B3	(50099)	STORY	Stores the byte in the accumulator on the video screen at the current cursor position.
\$FD25	(64805)	GOTKEY	If escape sequences are enabled, checks to see if ESC was pressed; if it was, then the keyboard is read again and any valid escape sequence is executed. Escape sequences are enabled if input is requested by the Applesoft INPUT command or the monitor's GETLN (\$FD35) subroutine. They are not enabled by the Applesoft GET command.

is active) and for redirecting the source of input to a diskette file if a ProDOS READ command is in effect. If a READ command is not in effect, then ProDOS uses the subroutine whose address is stored in its own input link to get input. The address stored here is initially that of one of the standard keyboard input subroutines.

If standard attempts are made to modify KSW, then ProDOS could be temporarily disconnected. With one exception, this means that you must not use any of the following methods to install a new input subroutine:

- Using an Applesoft "IN#s" command (as opposed to the ProDOS PRINT CHR\$(4); "IN#s" command) from within a program.
- Using Applesoft POKE commands to place new values directly into KSWL and KSWH.

- Using the Applesoft CALL command or the system monitor GO command (as opposed to the ProDOS BRUN command) to execute an assembly-language program that stores values directly into KSWL and KSWH.

The exception relates to the use of the BRUN command. If an assembly-language program is loaded and executed directly from diskette by using the BRUN command, then the program is permitted to modify the contents of KSW and both ProDOS and the new input subroutine will still remain active. This is because just before the program that is BRUN ends, ProDOS checks to see whether the input link has changed. If it has, it moves the link address into its own input link and places the address of its input subroutine back into KSW.

You can also successfully store a new input subroutine by storing its address directly into the ProDOS input links found at \$BE32 and \$BE33 instead of into KSW.

If you want to use an IN# command within an Applesoft program in order to redirect input to a particular port, you must use the ProDOS "version" of that command by printing a [control-D] character (ASCII code 4), immediately followed by "IN#s" (where "s" is the port number) and a carriage return. The [control-D] signifies to ProDOS that a ProDOS command is about to be presented; it can be generated using the Applesoft CHR\$ function. For example, to redirect input to serial port 2 when ProDOS is being used, execute the following statement from within a program:

```
PRINT CHR$(4);"IN#2"
```

instead of the Applesoft "IN#2" command. After this is done, both ProDOS and the new input subroutine will be active.

ProDOS supports a special form of the IN# command that its predecessor, DOS 3.3, does not. This special IN# command can be used to properly install an input subroutine that is located anywhere in memory and not just to pass control to a program located at a port. The only restriction on its use is that the first byte of the new input subroutine must be a 65C02 "CLD" (clear decimal flag) instruction. To install any such input subroutine, you must execute the statement

```
PRINT CHR$(4);"IN# Addr"
```

where "addr" represents either the decimal starting address of the new input subroutine or, if preceded by "\$", the hexadecimal starting address. For example, if your new input subroutine begins at \$300 (decimal 768), then you would execute either of the following two statements:

```
PRINT CHR$(4);"IN# A$300"
```

or

```
PRINT CHR$(4);"IN# A768"
```

and the new input subroutine will be properly installed in the ProDOS input link.

The Keyboard

The keyboard is one of the most important input/output device attached to the //c. It is one of two primary sources of input (the disk drive being the other one) and without it you would not be able to interact conveniently with any program running on the //c.

We are now going to take a close look at the keyboard. We will explain how it is used to enter information and present examples of how to modify the handling of keyboard input to meet special requirements.

Encoding of Keyboard Characters

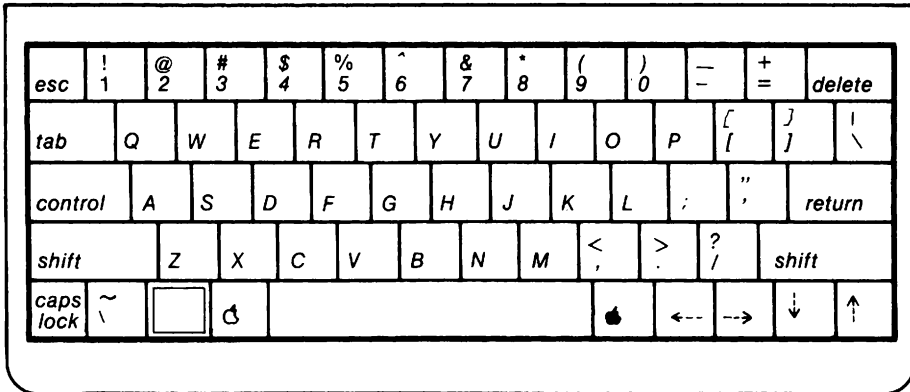
The //c's keyboard is made up of 62 typewriter-like keys that include most of the ones that you would see on a standard typewriter as well as a few more special ones. The keycaps are spatially arranged in the standard QWERTY (Sholes) configuration that is familiar to all typists. You should note, however, that as far as the //c is concerned, the mapping of keys to characters is not dictated by the keycaps, but rather by the setting of the "keyboard" switch which is found just above the keyboard to the right of the "reset" button and the "80/40" switch.

With the keyboard switch in its normal position (up), a QWERTY (Sholes) mapping scheme is used and the keycap does, indeed, indicate the character that is entered when a key is pressed. However, if the keyboard switch is down, then the keys are mapped to the Dvorak keyboard configuration shown in Figure 6-1. The characters on the Dvorak keyboard are arranged in such a way as to permit a typist to maximize typing speed. This is possible because the more frequently used letters are placed on the home row (where the fingers of a touch typist normally sit) so that they can be located and struck more rapidly than if they were located elsewhere. Conversely, the Sholes keyboard arrangement was designed to minimize typing speed so that the striking hammers in the original mechanical typewriters would not become tangled.

All of the keys on the keyboard except for the two "Apple" keys that flank the space bar, are used to generate the ASCII codes that the //c uses to represent the 52 alphabetic characters (26 uppercase and 26 lowercase), 10 digits (0 . . . 9), 34 special symbols, and 32 "control" codes that it recognizes.

Some keys on the keyboard do not generate ASCII codes when pressed by themselves, but are used to affect the code that is normally generated by another key that is pressed at the same time. These keys are the two SHIFT keys, the CONTROL key, and the CAPS LOCK key.

(a)



(b)

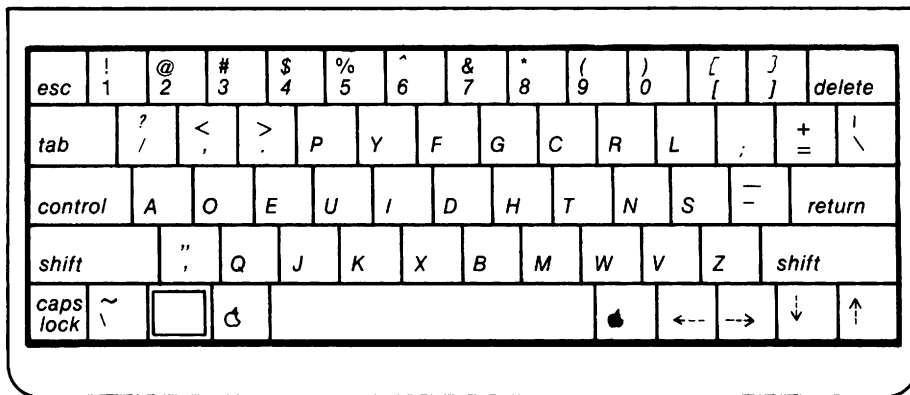


Figure 6-1. The Sholes and Dvorak keyboard layouts.

You can probably guess how the SHIFT keys affect the character codes already. Ignoring the effect of the CAPS LOCK key for the moment, if you press any alphabetic key by itself, you will generate an ASCII code for a lowercase character. If either SHIFT key is pressed at the same time, however, the ASCII code for the corresponding uppercase character is generated instead. The SHIFT key is also pressed to select the ASCII code for the top symbol on those keys that have two symbols marked on them.

The CAPS LOCK key, if in the down position, merely causes any lowercase alphabetic character code that is entered to be converted to the code for the corresponding uppercase character.

The CONTROL key acts in a similar way as the SHIFT keys. If you hold the CONTROL key down and then press any of the 26 alphabetic keys, then an ASCII code for a control character will be selected and not the code for the alphabetic key itself. (The remaining six control characters are generated by pressing the CONTROL key together with one of the following special symbols: @, [, \,], ^, and _).

Special Keys

There are several special keys on the //c's keyboard that you probably won't see on a standard typewriter. These are the ESC (for ESCape), TAB, DELETE, UP-ARROW, DOWN-ARROW, LEFT-ARROW, RIGHT-ARROW, OPEN-APPLE, and SOLID-APPLE keys as well as the CONTROL key that was discussed above.

The ESC, TAB, DELETE, and the four arrow keys all generate specific ASCII codes when they are pressed and they are often referred to as "editing" keys. Refer to Table 6-1 for the ASCII codes generated by these keys.

Different programs will perform different tasks when an editing key is pressed. It is hoped, however, that the tasks performed will relate in a meaningful way to the name or the symbol on the keycap. That is, it would be preferable if the ESC key actually caused you to ESCape (or exit) some part of a program and the TAB key caused the cursor to move several spaces to the right, and so on. It would be incredibly annoying, for example, if when you pressed the down-arrow key your cursor moved up or if you pressed the DELETE and the cursor moved five spaces to the right.

The "Apple" Keys

The OPEN-APPLE and SOLID-APPLE keys that flank the space bar are actually equivalent to push buttons #0 and #1, respectively, on a joystick or a pair of game controllers that are interfaced to the mouse/game connector at the back of the //c. These push buttons will be described in detail in Chapter 10. Although these keys cannot be used to generate ASCII codes, they could be used, with appropriate software, to act as special shift keys. The software, after reading a key, could check to see whether an "Apple" key was being pressed; if one was, then a different action could be taken than if the key were pressed by itself. For example, Apple has issued design guidelines urging software developers to consider the question mark key as a "HELP" key if it is pressed at the same time as the OPEN-APPLE key. Here is how you would implement a help function in an Applesoft program:

```
10 PRINT "Enter a command: ";GET A$
20 IF A$="?" AND PEEK(49249)>127 THEN 1000
```

```
1000 REM PLACE "HELP" CODE HERE
```

Memory location 49249 is the address of the location that holds the state of the OPEN-APPLE key. (49250 is used for the SOLID-APPLE key.) If the value read from this location is greater than 127 (that is, bit 7 is on), then OPEN-APPLE is being pressed.

The OPEN-APPLE key can also be used to modify the effect of resetting the //c. This is discussed in the last section of this chapter.

Keyboard I/O Locations

The Apple //c reserves two I/O memory locations for use by the keyboard I/O device. These two locations are \$C000 and \$C010 and their meanings are summarized in Table 6-6.

Table 6-6. Keyboard I/O locations.

<i>Address</i>		<i>Symbolic Name</i>	<i>Meaning</i>
<i>Hex</i>	<i>(Dec)</i>		
\$C000	(49152)	KBD	Keyboard data and strobe. Keyboard data is stored in bits 0 . . . 6. Bit 7 represents the keyboard strobe and will be 1 if keyboard data is ready to be read.
\$C010	(49168)	KBDSTRB or AKD	Clear keyboard strobe and read any-key-down status. Reading or writing this location will clear the keyboard strobe bit at \$C000. Bit 7 indicates whether a key is being pressed; if it is 1, then a key is being pressed.

KBD (\$C000) is used to hold the 7-bit ASCII code for any keyboard character that is entered as well as a 1-bit “strobe” flag. The strobe flag is held in bit 7 of KBD and indicates whether a key has been pressed and is ready to be presented to the system. If the bit is set to 1, then keyboard data is ready to be read; if it is cleared to 0, then no keyboard character has yet been entered since the last time the strobe was cleared. The lower 7 bits of KBD always contains the ASCII code of the last key entered. The keyboard strobe is also connected to an input line on the integrated circuit that controls serial port 2 at the back of the //c; this has been done in such a way that interrupts can be generated whenever a key is pressed. We will be looking at these keyboard interrupts in Chapter 11.

The second keyboard I/O location is KBDSTRB (\$C010). This location is used for two purposes. First, if any read or write operation is performed on this location, such as a PEEK or a POKE, then the keyboard strobe bit (in KBD) will be cleared to zero. This tells the //c’s built-in keyboard input subroutines that the keyboard data has already been dealt with and that no further information should be read from the keyboard until the strobe flag becomes set once again.

Second, bit 7 of KBDSTRB (\$C010), also called AKD (\$C010), indicates the status of the “any-key-down” flag. If it is 1, then a key is being pressed; if it is 0, then no key is being pressed. This flag is not the same as the strobe flag because, as we will see later on, there are times when even though a key is being pressed, it has not yet been “officially” strobed into the system.

Here is a simple assembly-language program to read data from the keyboard:

```
WAITFORKEY LDA $C000      ;Get keyboard data + strobe
            BPL WAITFORKEY ;Loop until strobe is set
            STA $C010      ;Clear keyboard strobe
```

The branch-on-plus (BPL) instruction will cause this program to loop until KBD becomes “negative,” that is, until bit 7 of KBD (the strobe bit) becomes 1.

In Applesoft, this program would be written as follows:

```
100 IF PEEK(49152)<128 THEN 100 : REM WAIT FOR STROBE
110 POKE 49168,0 : REM CLEAR KEYBOARD STROBE
```

It is important that the keyboard strobe be cleared after reading data from the keyboard. If it isn't, the program will keep thinking that a key has just been pressed whenever it checks for more keyboard data.

Let's look at a simple program, called TYPING.TIMER, that makes use of the AKD (\$C010) flag; it is shown in Table 6-7. This program analyzes your typing speed by displaying the length of time your finger stays on each key that you press and the time delay between successive keystrokes. It does this by simply monitoring the status of the AKD flag and keeping track of the elapsed time using a software counter. In a fully developed program of this sort, you would be able to quickly pinpoint a typist's problem areas.

After you enter the program, you can run it by entering CALL 768 from Applesoft direct mode. After you do this, type in four characters from the keyboard as fast as you can. After you have done this, a set of six times (in units of 20 microseconds) will be displayed. The meanings of each of these times are as follows:

First	:ON time for first key
Second	:delay between first and second keys
Third	:ON time for second key
Fourth	:delay between second and third keys
Fifth	:ON time for third key
Sixth	:delay between third and fourth keys

To convert the displayed numbers into milliseconds, simply divide them by fifty. You should take note of how the decimal values for these times are displayed. The program makes use of an Applesoft subroutine called LINPRT (\$ED24); this subroutine takes a binary number that is in X (low byte) and A (high byte) and displays it as an unsigned decimal number.

Table 6-7. TYPING.TIMER—a program to measure your typing speed.

```

1  *****
2  * TYPING.TIMER *
3  *****
4
5  CHARS EQU 3           ;Number of chars. to be typed
6
7  AKD EQU $C010         ;Any-key-down flag
8
9  HEXDEC EQU $ED24      ;Hex-to-decimal conversion
10 CRUT EQU $FD8E        ;Send a CR
11
12 ORG $300
13
14 LDX #0
15 JSR RELEASE
16 JSR PRESS
17 INX
18 INX
19 INX
20 INX
21 CPX #CHARS*4
22 BNE NEXTKEY
23
24 * Display the results:
25 LDY #0
26 TIMEDSP LDA TIMEON,Y
27 TAX
28 LDA TIMEON+1,Y
29 STY YSAVE
30 JSR HEXDEC
31 JSR CRUT

```

0300: A2 00
0302: 20 41 03
0305: 20 2C 03
0308: E8
0309: E8
030A: E8
030B: E8
030C: E0 0C
030E: D0 F5

0310: A0 00
0312: B9 58 03
0315: AA
0316: B9 59 03
0319: 8C 57 03
031C: 20 24 ED
031F: 20 8E FD

;Get a keystroke

;Display in decimal

0322: AC 57 03	LDY	YSAVE	
0325: C8	INY		
0326: C8	INY		
0327: C0 0C	CPY	#CHARS*4	
0329: D0 E7	BNE	TIMEDSP	
032B: 60	RTS		
032C: A9 00	time	is ~20 microsec.)	
032E: 9D 59 03	PRESS	LDA #0	
0331: 9D 58 03		STA TIMEON+1,X	; Initialize "ON" timer
0334: FE 58 03		STA TIMEON,X	
0337: D0 03	KEYWAIT	INC TIMEON,X	; Bump the time count
0339: FE 59 03		BNE KEYWAIT1	
033C: 2C 10 C0	KEYWAIT1	INC TIMEON+1,X	
033F: 30 F3		BIT AKD	; Is key still pressed?
		BMI KEYWAIT	; Yes, so wait
0341: A9 00	RELEASE	LDA #0	
0343: 9D 5B 03		STA TIMEOFF+1,X	; Initialize "OFF" timer
0346: 9D 5A 03		STA TIMEOFF,X	
0349: FE 5A 03	RELWAIT	INC TIMEOFF,X	; Bump the time count
034C: D0 03		BNE RELWAIT1	
034E: FE 5B 03		INC TIMEOFF+1,X	
0351: 2C 10 C0	RELWAIT1	BIT AKD	; Has key been released?
0354: 10 F3		BPL RELWAIT	; No, so wait
0356: 60	RTS		
YSAVE	DS	1	
TIMEON	DS	2	; Duration of keypress
TIMEOFF	DS	2	; Duration of key release
	DS	CHARS*4-4	; Data for other keystrokes

Modifying the Keyboard Input Subroutine

Earlier in this chapter, we saw how it was possible to replace the subroutine that the //c uses in order to obtain character input by simply changing the input link at KSW (\$38/\$39). At that time, we indicated that it would be possible to install a wide variety of subroutines that would still obtain input from the keyboard but would do it in different, more useful, ways.

Look at the program called MACRO.ENTRY in Table 6-8. It must be installed by using the BRUN command to execute it directly from diskette. This program allows you to automatically enter a commonly used command phrase from the keyboard simply by pressing the OPEN-APPLE key at the same time as one of three other keys, C, H, or L. These keys will generate the following sequences of characters:

C → CATALOG,D1 (followed by [return])
H → HOME (followed by [return])
L → LOAD (followed by [space])

A key that is used to enter a whole string of other characters is called a macro key. With MACRO.ENTRY installed, it is a simple matter to catalog the disk, clear the screen, or to “type in” LOAD before specifying the name of a program. All you must do is press OPEN-APPLE and the appropriate macro key.

The first part of MACRO.ENTRY simply sets up the new input link so that it points to NEWIN, the start of the new input routine. This means that every time a program requests input from the //c by calling the standard RDKEY (\$FD0C) input subroutine, control will eventually pass to NEWIN instead of the standard keyboard input subroutine.

When NEWIN is entered, some registers that will be used are first saved and then a location (called MACROFLG) is checked to see whether a macro entry is currently being processed. If not, the program enters a tight loop until a keypress is detected. After a key has been pressed, the character is loaded into the accumulator and the status of the OPEN-APPLE key is examined with a BIT OPENAPL instruction. If it is not being pressed, then the following BPL instruction will succeed (because bit 7 of OPENAPL will be 0) and control will return to the calling program as usual.

However, if OPEN-APPLE is being pressed, then the MACROKEY table is scanned to see whether it contains the keyboard character that has been entered. If it doesn't, then control returns to the calling program. If it does, then the high-order bit of MACROFLG is set and the first character of the entry in the PHRASES table is returned to the calling program. Each time that input is requested after this, the next character in the macro phrase will be returned to the calling program. This will continue until all characters have been returned, at which time MACROFLG is cleared.

Table 6-8. MACRO.ENTRY—a program to define macro keys.

```

1  *****
2  * MACRO.ENTRY *
3  *****
4
5  * (BRUN this program from disk)
6
7  MTOTAL EQU 3           ;Number of macro keys in MACROKEY
8
9  KSW EQU $38           ;Input "link"
10
11 OPENAPL EQU $C061      ;OPEN-APPLE switch
12 SHOWCUR EQU $CC4C
13 UPDATE EQU $CC70
14 GOTKEY EQU $FD25      ;Handle ESC sequence
15
16 DRG $300
17
18 * Set up new input link:
19
20 LDA #<NEWIN
21 STA KSW
22 LDA #>NEWIN
23 STA KSW+1
24 RTS
25
26 * This is the new input routine:
27
28 NEWIN STX XSAVE
29 STY YSAVE
30 BIT MACROFLG          ;Are we processing a macro?
31 BMI GETMAC            ;Yes, so branch

```

(continued)

Table 6-8. MACRO ENTRY—a program to define macro keys (continued).

0314:	20	4C	CC	32	NEWIN2	JSR	SHOWCUR	
0317:	20	70	CC	33	NEWIN3	JSR	UPDATE	;Look for key
031A:	10	FB		34		BPL	NEWIN3	
031C:	2C	61	C0	35		BIT	OPENAPL	;Is OPEN-APPLE being pressed?
031F:	10	4C		36		BPL	EXIT	;No, so exit
0321:	A2	00		37		LDX	#0	
0323:	DD	79	03	38	ORIGSCAN	CMP	MACROKEY,X	;Is this a command key?
0326:	F0	07		39		BEQ	FINDMAC	;Yes, so branch
0328:	E8			40		INX		;Go on to next item in table
0329:	E0	03		41		CPX	#MTOTAL	;At end of table?
032B:	D0	F6		42		BNE	ORIGSCAN	;No, so keep looking
032D:	F0	3E		43		BEQ	EXIT	
032F:	A9	80		44	FINDMAC	LDA	#\$00	
0331:	8D	77	03	45		STA	MACROFLG	;Set "macro in effect" flag
0334:	8E	76	03	46		STX	CMDNUM	
0337:	A2	00		47		LDX	#0	
0339:	8E	78	03	48		STX	MACROPOS	
033C:	A0	00		49		LDY	#0	
033E:	CC	76	03	50	FINDMAC1	CPY	CMDNUM	;Have we found the macro?
0341:	F0	13		51		BEQ	GETMAC	;Yes, so branch
0343:	AE	78	03	52	SKIPMAC	LDX	MACROPOS	
0346:	BD	7C	03	53		LDA	PHRASES,X	;Get macro character
0349:	F0	05		54		BEQ	FINDMAC2	;Branch if past end
034B:	EE	78	03	55		INC	MACROPOS	; else move to next position
034E:	D0	F3		56		BNE	SKIPMAC	
0350:	EE	78	03	57	FINDMAC2	INC	MACROPOS	
0353:	C8			58		INY		;Increment macro count
0354:	D0	E8		59		BNE	FINDMAC1	
0356:	AE	78	03	60	GETMAC	LDX	MACROPOS	
0359:	BD	7C	03	61		LDA	PHRASES,X	;Get new character
035C:	EE	78	03	62		INC	MACROPOS	;Update position within macro
035F:	C9	00		63		CMP	#0	;At the end?
0361:	D0	07		64		BNE	EXIT1	;No, so exit

```

0363: A9 00          LDA #0
0365: 8D 77 03        STA MACROFLG ;Clear "macro in effect" flag
0368: F0 AA            BEQ NEWIN2    ; and get a keystroke
036A: AC 75 03        LDY YSAVE
036D: AE 74 03        LDX XSAVE
0370: 20 25 FD        JSR GOTKEY    ;Handle ESC sequence
0373: 60              RTS

71
72
73 XSAVE DS 1
74 YSAVE DS 1
75 CMDNUM DS 1
76 MACROFLG DFB 0 ;0=no macro / $80=macro
77 MACROPOS DFB 0
78

79 * Table of macro keys:
80 * (high bit must be on)
81 MACROKEY ASC "C"
82 ASC "H"
83 ASC "L"
84

85 * Table of macro phrases:
86 * (each entry must end with a 0)

87 PHRASES ASC "CATALOG,D1"
88 DFB $8D,0
89 ASC "HOME"
90 DFB $8D,0
91 ASC "LOAD "
92 DFB 0

0377: 00
0378: 00

0379: C3
037A: C8
037B: CC

037C: C3 C1 D4
037F: C1 CC CF
0382: C7 AC C4
0385: B1
0386: 8D 00
0388: C8 CF CD
038B: C5
038C: 8D 00
038E: CC CF C1
0391: C4 A0
0393: 00

```

If you want to change the macro commands and associated entries, then you must modify the **MACROKEY** and **PHRASES** tables. The ASCII code for each command key must be stored in the **MACROKEY** table with the high bit on. The corresponding macro phrases for each key must be stored, in order, in the **PHRASES** table; each phrase must be terminated by a **00** byte. In addition, you must set **MTOTAL** equal to the number of macro keys in the **MACROKEY** table before reassembling the program.

Recall that only locations **\$300** through **\$3CF** are available for use in page three of memory. You must ensure that your macro tables are short enough that you do not spill over the **\$3CF** boundary.

Keyboard Auto-Repeat

When you enter a key that corresponds to a particular ASCII code, that code will begin to repeat after you have kept the key pressed for longer than about **900** milliseconds. (This time could be shorter, but heavy-handed typists might then encounter difficulties.) Once this “pre-repeat” period has elapsed, the code will begin to repeat itself 15 times per second (that is, once every **66.7** milliseconds). The auto-repeat phenomenon is generated by circuitry on the //c’s motherboard.

The auto-repeat feature is useful if you are editing programs or if are you are using word-processing programs. In both cases, it is often necessary to repeat character sequences or use an arrow key several times in succession to move the cursor to a new position. These tasks can be done easily merely by pressing the appropriate key and holding it down until the key is repeated as many times as is required.

The timing diagram for the keyboard’s auto-repeat function is shown in Figure 6-2. As soon as a key is first pressed, the **AKD (\$C010)** flag is turned on and, a few microseconds later, the keyboard strobe is turned on. The keyboard strobe will then stay on until it is cleared by accessing **KBDSTRB (\$C010)**. This is done right after the keyboard is read by the standard keyboard input subroutines. Note, however, that if the key is still being pressed, the **AKD** flag will remain on even after the strobe has been cleared.

After the strobe is cleared, and if the key is still being pressed, there is a short delay of about **900** milliseconds (called the “pre-repeat” delay) and then the keyboard strobe is turned on again. As usual, it will stay on until **KBDSTRB** is accessed once again. The width of the strobe pulse will depend on how rapidly the strobe is cleared after the strobe is turned on. Figure 6-2 was prepared by assuming that this is happening soon after the strobe is high and certainly much faster than the rate at which the key repeats.

At this stage, the keyboard strobe will automatically be turned on once every **66.7** milliseconds after it has been cleared and until the key is finally released. Even while the keyboard strobe is being turned on and off, however, the **AKD** flag remains on; in fact, **AKD** is turned off only when the key is

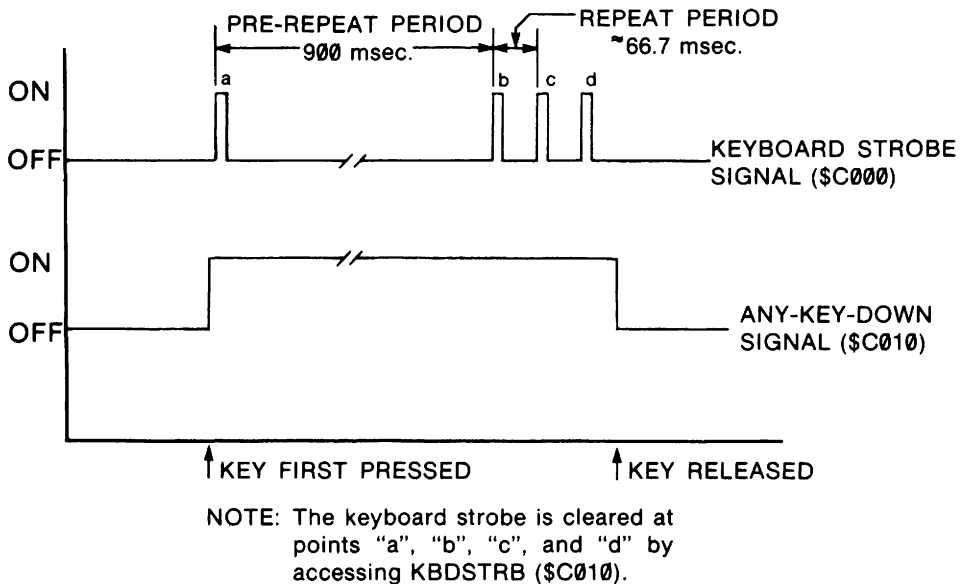


Figure 6-2. Keyboard auto-repeat timing diagram.

finally released. Thus, there are substantial periods of time when even though the AKD flag is on (that is, a key is being pressed), the keyboard strobe is not on.

Since most keyboard input subroutines, including the standard ones used in the `//c`, rely on the keyboard strobe to detect the presence or absence of a valid key code, the key code will be repeated at the same rate that the strobe is turned on (this is fixed by the `//c`'s internal circuitry). If, however, an alternative input subroutine is used that examines the AKD flag and returns a key code if it is on continuously for a given time period (even though, at the end of the period, the keyboard strobe may not be on), then a different repeat rate can be generated in software.

The `AUTO.REPEAT` program in Table 6-9 shows you how to adjust the auto-repeat rate in software. This program must be installed by using the `BRUN` command to load and execute it directly from diskette. `AUTO.REPEAT` modifies the input link so that it points to the code beginning at `NEWIN`. After this has been done, all requests for keyboard input will be processed by this subroutine and a much faster auto-repeat rate will be observed.

The first thing the new input subroutine does when it is called is to determine whether a new character was entered the last time it was called (`RPTFLAG` = `$00`) or whether an old one was being repeated (`RPTFLAG` = `$80`). If a new character was entered, then the accumulator is loaded with the number given by `PREDELAY` (the pre-repeat delay time); if not, it is loaded with the smaller number given by `RPTDELAY` (the auto-repeat time interval).

Table 6-9. AUTO.REPEAT—a program to speed up the auto-repeat rate of the keyboard.

```

1  *****
2  * AUTO.REPEAT *
3  *****
4
5  * (BRUN this program from disk)
6
7  PREDELAY EQU 150      ;Delay before repeating begins
8  RPTDELAY EQU 20       ;Delay between repeats
9
10 KSW EQU $38           ;Input "link"
11 CURSOR EQU $7FB       ;Cursor to use
12
13 KBD EQU $C000         ;Keyboard data + strobe
14 KBDSTRB EQU $C010     ;Clear keyboard strobe
15 AKD EQU $C010        ;Any-key-down flag
16
17 STORY EQU $C3B3       ;Store char at cursor position
18 SHOWCUR EQU $CC4C     ;Display cursor
19 GOTKEY EQU $FD25      ;Handle ESC sequences
20
21 ORG $300
22
23 * Install new input subroutine:
24
25 LDA #<NEWIN
26 STA KSW
27 LDA #>NEWIN
28 STA KSW+1
29 LDA #0
30 STA CURSOR

```

0300: A9 0E
0302: 85 38
0304: A9 03
0306: 85 39
0308: A9 00
030A: 8D FB 07

Use non-flashing cursor


```

030D: 60      31      RTS
030E: 48      32      NEWIN
030F: 8C 66 03 33      PHA
0312: 20 4C CC 34      STY YSAVE
                                35      JSR SHOWCUR
                                36      ;Save screen character
                                37      ;Save Y-register
                                38      ;Set up cursor
                                39      * Wait before repeating:
0315: A9 14 38      LDA #RPTDELAY
0317: 2C 65 03 39      BIT RPTFLAG
031A: 30 02 40      BMI WAIT
031C: A9 96 41      LDA #PREDELAY
031E: 38 42      SEC
031F: A0 80 43      LDY #128
0321: 2C 10 C0 44      BIT AKD
0324: 10 20 45      BPL RPTOFF
0326: 88 46      DEY
0327: D0 F8 47      BNE WAIT2
0329: E9 01 48      SBC #1
032B: D0 F2 49      BNE WAIT1
                                50
                                51
                                52      * If we've reached here, we are repeating (unless another
                                53      * key was pressed before releasing the first one). The
                                54      * following code reads the keyboard (before its code
                                55      * has been strobed in) and sets the high bit as per
                                56      * the standard input protocol:
                                57
032D: AD 00 C0 58      LDA KBD
0330: 2C 10 C0 59      BIT KBDSTRB
0333: 09 80 60      ORA #$80
0335: CD 64 03 61      CMP OLDKEY
0338: F0 04 62      BEQ RPTON
033A: A0 00 63      LDY #0
                                64
                                65      ;Get key code
                                66      ;Clear strobe (just in case)
                                67      ;Set high bit
                                68      ;Same as previous key?
                                69      ;Yes, so we're repeating
                                70      ;Repeat off

```

(continued)

Table 6-9. AUTO.REPEAT—a program (continued)

033C: F0 02	64	BEQ FIXRPT	(always taken)
033E: A0 80	65	LDY #80	;Repeat on
0340: 8C 65 03	66	STY RPTFLAG	;Adjust the repeat flag
0343: 4C 53 03	67	JMP GETKEY1	
	68		
	69	* Key was lifted, so wait for standard keypress:	
	70		
0346: A9 00	71	LDA #0	
0348: 8D 65 03	72	STA RPTFLAG	;Repeat off
	73		
034B: AD 00 C0	74	LDA KBD	;Has a key been strobed in?
034E: 10 FB	75	BPL GETKEY	;No, so branch
0350: 2C 10 C0	76	BIT KBDSTRB	;Clear keyboard strobe
0353: 8D 64 03	77	STA OLDKEY	;Save key code for next time
	78		
0356: 68	79	PLA	;Get old screen character
0357: AC 66 03	80	LDY YSAVE	;Restore Y-register
035A: 20 B3 C3	81	JSR STORY	;Restore screen char.
035D: AD 64 03	82	LDA OLDKEY	;Get the key code
0360: 20 25 FD	83	JSR GOTKEY	;Handle ESC sequences
0363: 60	84	RTS	
	85		
0364: 00	86	OLDKEY DFB 0	;Last key pressed
0365: 00	87	RPTFLAG DFB 0	;0=not repeating / \$80=repeating
	88	YSAVE DS 1	;Temporary storage area for Y

A delay loop is then entered during which the status of the AKD flag is repeatedly checked. If the flag is turned off (that is, the key is released) at any time before the loop finishes, then RPTFLAG is set equal to \$00 (to indicate that repeating has ended) and then keyboard input is requested in the standard way (by waiting for the keyboard strobe to be turned on). After a keyboard character is received, it is stored in OLDKEY.

If a key remains pressed until the timing loop finishes, then the keyboard data is immediately read from KBD (\$C000), even though the strobe may not actually be on. This data will usually equal the code stored in OLDKEY (the previous key strobed in). If at some time during the loop, however, another key was pressed before the previous one was released, it will be different. If the key code is the same (the usual case), RPTFLAG is set equal to \$80. This indicates that an auto-repeat sequence is in effect so that the next time input is requested, the shorter "RPTDELAY" delay loop will be selected. Otherwise, RPTFLAG is set to \$00. In either case, the key code is stored in OLDKEY before the subroutine finishes.

The important point to note here is that the keyboard data will always be read after the key has been pressed for the length of time set by the loop counter (PREDELAY or RPTDELAY). Thus, we can select both the auto-repeat rate and the predelay time simply by changing the RPTDELAY and PREDELAY constants. You may want to try out different repeat rates and predelay times by changing these constants in the program. Be warned, however, that if you set RPTDELAY too low, your reflexes may not be fast enough to control the speeding cursor! You should also be careful not to set PREDELAY too low or else you may not be able to press and release a key before it starts to repeat!

Resetting the Apple //c

The RESET button is located above the upper left-hand corner of the keyboard just above the ESC key. It should really be called a "panic" button since it is usually used to interrupt the running of a program when all else fails. After a reset signal is generated by pressing the RESET button, the //c generally returns to Applesoft direct mode from where you can easily examine the program that was just running or you can load and run another one.

Actually, the RESET button does nothing really important if you press it by itself. If you press it while you are also holding down the CONTROL key, however, then the reset pin on the 65C02 microprocessor will be held in a low state, causing the 65C02 to begin its standard reset procedure. This procedure was described in Chapter 2.

Special RESET Procedure

If the OPEN-APPLE key is held down when [control-RESET] is pressed, then a "cold" reset procedure will begin that always allows you to restart the

//c. This means that the diskette in the //c's built-in disk drive will start to boot up just as it did when the power was first turned on. This reset procedure cannot be prevented and will destroy any program that may be in memory when it is requested.

Trapping "Soft" RESETs

The reset procedure just mentioned cannot be avoided using software techniques because it is wholly contained within the //c's ROM. It is possible, however, to redirect a standard "soft" reset (invoked by pressing [control-RESET] by itself) to any routine that you want to use to trap such a condition.

When [control-RESET] is pressed, the 65C02 microprocessor jumps to a location stored at locations \$FFFC/\$FFFD (low byte first). On the //c, these locations are stored either in the system monitor ROM area or in the bank-switched RAM area that shares the same address space as the system monitor, depending on which one was enabled when the reset signal occurred. (See Chapter 8 for a discussion of bank-switched RAM.) The ROM locations always hold the address of RESET (\$FA62) in the system monitor but any address can be stored in bank-switched RAM. If ProDOS is being used, the address stored is \$FFCB, which is the start of a ProDOS subroutine that re-enables the system monitor ROM and then passes control to the standard reset handler at RESET (\$FA62).

The subroutine that begins at \$FA62 takes care of some general-purpose housekeeping chores (like setting normal video, selecting the keyboard and video screen for I/O, and so on), checks for the special OPEN-APPLE reset, and then, if the special reset has not been requested, checks to see whether a user-defined reset handling routine should be executed.

If the result of the logical exclusive-OR of the value stored at SOFTEVH (\$3F3) with the constant \$A5 is stored at PWREDUP (\$3F4), the //c will jump to a subroutine whose address is tained in SOFTEV (\$3F2/\$3F3) (low-order byte first). If this test fails, then the the diskette in the disk drive will be rebooted. The important reset locations are summarized in Table 6-10.

Thus, to trap a RESET condition, two things must be done:

- The address of the subroutine that is to take control after a reset must be stored at SOFTEV.
- The byte stored at \$3F3 must be logically exclusive-ORed with \$A5, and the result stored at \$3F4. This can be done by executing the following instructions:

```
LDA $3F3
EOR #$A5
STA $3F4
```

Right after the //c is turned on, SOFTEV is initialized to \$E000, the cold

Table 6-10. Reset interrupt locations.

<i>Address</i>		<i>Symbolic Name</i>		<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>			
\$FFFC	(65532)	RESETV	(low)	Reset vector. These locations contain the address of the subroutine that is called when a reset signal occurs.
\$FFFD	(65533)		(high)	
\$03F2	(1010)	SOFTEV	(low)	User-defined reset vector. These locations contain the address of a user-installed subroutine to which control is passed when a reset signal occurs (if PWREDUP is set up properly).
\$03F3	(1011)		(high)	
\$3F4	(1012)	PWREDUP		Powered-up byte. If the value stored here is the same as the result of the logical exclusive-OR of the value at \$3F3 with \$A5, then control will pass to the user-defined subroutine specified by SOFTEV.

start for Applesoft, and then, after [control-RESET] has been pressed once, to \$E003, the warm start for Applesoft. If ProDOS is being used, then SOFTEV will later be adjusted so that it points to a reset handler within ProDOS itself. This handler takes care of reconnecting ProDOS after RESET is pressed and of entering Applesoft direct mode. (When RESET is pressed, the addresses of the standard keyboard input and output subroutines are stored in the input and output links, thus effectively “removing” ProDOS from the system.)

By the way, the reason for storing the “funny” number at PWREDUP is to allow the //c to detect whether or not it has just been turned on. If it has been, then it is highly unlikely that PWREDUP will be properly “related” to SOFTEVH, and the //c will interpret this to mean that the diskette must be automatically booted.

Trapping RESET from Assembly Language

If an assembly-language program is being executed, any reset condition that may occur can be easily trapped by setting up SOFTEV and PWREDUP as indicated above as soon as the program begins. The error-handling routine to which SOFTEV points must handle the reset in an orderly manner; its main duty will be to ensure that the data areas of the program still make sense and to take appropriate action if they do not. For example, if RESET is pressed after one byte of a two-byte pointer has been set up, then the reset

handler had better detect this and fix it or else the next time that the program uses this (incomplete) pointer it will disappear into outer space.

It is also important to ensure that the reset-handling routine adjusts the stack pointer to a suitable value. Remember that RESET can be pressed at any time, including times when there are several bytes of information stored on the stack. If you don't adjust the stack pointer downward in these situations, it might eventually "overflow," allowing you to overwrite important information stored on the stack. A simple way of handling this problem is to always reset the stack pointer to its value when the program was first entered. To do this, execute the following two instructions when beginning your assembly-language program:

```
TSX
STX STACKSV
```

where STACKSV refers to a memory location. In the reset-handling routine, the original stack pointer can be restored by executing the following instructions:

```
LDX STACKSV
TXS
```

and then the remainder of the reset-handling routine can be executed.

Another important chore for the reset-handling routine to perform is to reconnect ProDOS (recall that ProDOS is deactivated whenever the //c is reset). This is most easily done by initially saving the ProDOS addresses stored in the input and output links (at \$36 . . . \$39) in safe locations so that the links to ProDOS can be restored when RESET is trapped.

Trapping RESET from Applesoft

RESET can be trapped while running an Applesoft program by using Applesoft's built-in error-handling subroutine. If this is done properly, then every time the //c is reset, the program can be caused to go to the line number that is specified in the currently active ONERR GOTO statement.

The following steps must be performed by the subroutine that traps reset when Applesoft is active:

- The ProDOS I/O addresses must be stored in the I/O links to reactivate ProDOS.
- The value stored at VFACTV (\$67B) must be set equal to the value stored there before the //c was reset. VFACTV is used as a flag by the video firmware to determine whether the 80-column firmware is to be used to perform video operations. If VFACTV is greater than 127, then standard 40-column mode is active; otherwise, the 80-column firmware is active.
- The 80-column video display must be turned on (but only if it was on before the //c was reset).

- A subroutine at \$D683 must be called to properly configure the 65C02 stack.
- The program should put an error code number in the X register and then execute a "JMP \$D412" to pass control to the Applesoft ONERR GOTO handler. (The handler will place the error code number in location 222.)

To install such a subroutine, its starting address must be stored in SOFTEV (low-order byte first) and PWREDUP must be properly adjusted as discussed above.

You can reactivate ProDOS using the technique mentioned at the end of the last section: save the values of the I/O links when the reset-handling subroutine is first installed and then restore them when RESET is pressed.

When RESET is pressed, the //c automatically turns off the 80-column display and a \$00 byte is stored at VFACTV (\$67B) to indicate that the 80-column firmware is not in use. Thus, the reset-trapping subroutine must take care of restoring VFACTV to its value just before RESET was pressed and of re-enabling the 80-column display, if necessary, by writing to 80COLON (\$C00D). Note, however, that this last step is only to be performed if the 80-column display was active when RESET was pressed. The only way for the reset handler to determine whether it was is to examine a flag that contains the contents of the 80COL (\$C01F) status location immediately before the //c was reset. This flag can be initialized in the subroutine that sets up the reset vector, but it must be updated whenever the display mode is changed. Similarly, the value of VFACTV must be stored in a safe temporary location so that it can be restored properly after RESET has been pressed.

The subroutine at \$D683 must be called in order to fix a bug in Applesoft which arises when an error (that is not handled by the Applesoft RESUME command) occurs within a FOR/NEXT loop or a GOSUB/RETURN subroutine. This bug causes incorrect information to be left on the 65C02 stack after the error is processed.

Applesoft and ProDOS all store error code numbers in location \$DE (222) whenever an error occurs. The ONERR GOTO error-handling routine can then examine this location using a PEEK(222) command to determine what kind of error occurred. (See Table 6-12 for a list of Applesoft and ProDOS error code numbers.) Several error code numbers are not used by either Applesoft or ProDOS, including number 253. Thus, if the X register is loaded with 253 just before calling \$D412, the Applesoft error-handling subroutine will be able to detect a reset "error" by determining whether PEEK(222) = 253. PEEK(222) = 253.

Table 6-11 contains a program that will set up the reset vector so that it points to a subroutine that traps reset when an Applesoft program is being run. To use it effectively, an ONERR GOTO statement must always be active; that is, error-trapping should never be turned off with a POKE 216,0 com-


```

031B: A5 38      LDA KSW
031D: 8D 5A 03   STA KSWTEMP
0320: A5 39      LDA KSW+1
0322: 8D 5B 03   STA KSWTEMP+1
0325: A5 36      LDA CSW
0327: 8D 5C 03   STA CSWTEMP
032A: A5 37      LDA CSW+1
032C: 8D 5D 03   STA CSWTEMP+1
032F: 60        RTS

41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

0330: AD 5A 03   LDA KSWTEMP
0333: 85 38      STA KSW
0335: AD 5B 03   LDA KSWTEMP+1
0338: 85 39      STA KSW+1
033A: AD 5C 03   LDA CSWTEMP
033D: 85 36      STA CSW
033F: AD 5D 03   LDA CSWTEMP+1
0342: 85 37      STA CSW+1

52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

0344: AD 5F 03   LDA VFTEMP
0347: 8D 7B 06   STA VFACTV
034A: 2C 5E 03   BIT FLAG80
034D: 10 03      BPL ERRHNDL1
034F: 8D 0D C0   STA COL800N

0352: 20 83 D6   JSR FIXSTACK
0355: A2 FD      LDX #253
0357: 4C 12 D4   JMP ERRFN

62
63
64
65
66

KSWTEMP DS 2
CSWTEMP DS 2
FLAG80 DS 1
VFTEMP DS 1

```

;Save current DOS links
 ;Restore DOS input link
 ;Restore DOS output link
 ;Restore VFACTV flag
 ;Was 80-column mode active?
 ;No, so branch
 ;Yes, so turn it on again
 ;Fix stack bug
 ;RESET = error code #253
 ;Go to ONERRGOTO handler
 ;DOS input hook
 ;DOS output hook
 ;>=\$80 if 80-column mode
 ;VFACTV before RESET

Table 6-12. Applesoft and ProDOS error codes and messages.

<i>Error Code</i>	<i>Error Message</i>
0	no error occurred [ProDos] or NEXT WITHOUT FOR [Applesoft]
2	RANGE ERROR
3	NO DEVICE CONNECTED
4	WRITE PROTECTED
5	END OF DATA
6	PATH NOT FOUND
7	PATH NOT FOUND
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	INVALID PARAMETER
12	NO BUFFERS AVAILABLE
13	FILE TYPE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	SYNTAX ERROR
17	DIRECTORY FULL
18	FILE NOT OPEN
19	DUPLICATE FILE NAME
20	FILE BUSY
21	FILE(S) STILL OPEN
22	DIRECT COMMAND [ProDOS] or RETURN WITHOUT GOSUB [Applesoft]
42	OUT OF DATA
53	ILLEGAL QUANTITY
69	OVERFLOW
77	OUT OF MEMORY
90	UNDEFINED STATEMENT
107	BAD SUBSCRIPT
120	REDIMENSIONED ARRAY
133	DIVISION BY ZERO
163	TYPE MISMATCH
176	STRING TOO LONG
191	FORMULA TOO COMPLEX
224	UNDEFINED FUNCTION
254	BAD RESPONSE TO INPUT STATEMENT
255	[control-C] PRESSED

mand. The error-trapping subroutine must be installed by loading into memory and then executing a CALL 768 command. It must not be BRUN directly from diskette.

Further Reading for Chapter 6

On changing the input link . . .

G. Little, "Zoom and Squeeze", *Micro*, July 1980, pp. 37–38. This article shows how the input link can be changed to control keyboard input.

On ProDOS and the input link . . .

C. Fretwell, "Setting I/O Hooks in ProDOS", *Call -A.P.P.L.E.*, April 1984, p.39

On trapping RESET . . .

E.A. Seiden, "ProDOS Reset Trap", *Nibble*, October 1984, p.107. This article presents an alternative way to trap RESET when ProDOS is active.

7

Character and Graphic Output and Video Display Modes

In this chapter, we will be discussing the primary output device supported by the `//c`: the video display monitor. This will include an analysis of how both text and graphic information are generated and how alternative output devices can be easily integrated into the system.

The `//c` is capable of controlling the video display in such a way as to support three general classes of output modes:

- Text mode
- Low-resolution graphics mode
- High-resolution graphics mode

All of these modes can exist in a standard single-width format or in a special double-width format (which includes an 80-column text mode).

Text mode is used whenever the standard character symbols represented by ASCII codes or the special limited graphics character set called `MouseText` are to be displayed on the screen. Both graphics modes are used to illuminate distinct blocks (low-resolution graphics) or points (high-resolution graphics) on the screen so that a variety of shapes can be generated. Whereas text can be displayed in black-and-white only, both graphics modes can be used to generate colored images if a color monitor or television set is connected to the `//c`.

The `//c` uses a “memory-mapped” video display technique. This means that the display of information on the video screen can be controlled simply by storing bytes of information in special memory locations that make up part of the 65C02’s 64K address space; these locations are mapped to unique positions on the screen display. Similarly, information shown on the video screen can be retrieved by reading the contents of these memory locations. These memory locations are connected to the `//c`’s video display support

circuitry in such a way that the bytes stored are converted into appropriate pictorial representations on the video display monitor.

Text Mode

The //c is capable of displaying text in either a 40-column-by-24-row mode or a double-width 80-column-by-24-row mode.

There are actually two versions of 40-column text mode supported by the //c. The “standard” 40-column mode is easily identified by its characteristic “checkerboard” cursor that is displayed whenever keyboard information is being requested. The other version is the “special” 40-column mode that is available when the //c’s 80-column firmware is being used; in this mode the cursor is an inverse block that does not flash.

The 80-column text mode can be invoked in several ways. The most common are as follows:

- Entering a PR#3 command from Applesoft direct mode
- Executing a PRINT CHR\$(4);“PR#3” command from within an Applesoft program
- Entering the ESC 8 escape sequence whenever the //c is requesting a line of information

All of these commands activate the //c’s special 80-column firmware which is capable of displaying information properly on the 80-column text screen. This is done by changing the addresses stored in the //c’s input and output links and setting various flags in memory locations that are examined by the system monitor’s subroutines.

Once you are in 80-column mode, you can switch between the 80-column mode and the special 40-column mode whenever keyboard information is being requested by using the two special escape sequences discussed in Chapter 6: ESC 4 and ESC 8. For example, if you are in 80-column mode and you want to enter the special 40-column mode, enter the sequence

ESC 4

If you want to go in the opposite direction, enter the sequence

ESC 8

To leave either the special 40-column mode or 80-column mode and go back to the standard 40-column mode, you can do one of two things (apart from resetting the system). First, you can enter ESC [control-Q] from the keyboard or you can print a [control-U] character (by using a PRINT CHR\$(21) command).

If you are not using ProDOS, you can also use the PR#0 and IN#0 commands to reconnect the standard 40-column input and output subroutines,

KEYIN (\$FD1B) and COUT1 (\$FDF0), and to turn off the 80-column screen display. ProDOS will ignore these commands, however, for the sake of compatibility with the Apple IIe. (On the Apple IIe, entering PR#0 when the 80-column display is enabled will reconnect the standard 40-column I/O links alright, but will not turn off the 80-column display. Not a pretty sight!)

In the following sections, we will be taking a closer look at the memory-mapped video RAM areas, the standard character output subroutines that are built into the IIc, and the soft switches used to select the video display mode.

The 80/40 Switch

Before we go on, a word about the “80/40” switch that is located just above the keyboard to the right of the “reset” button. Despite its name, this switch has no effect at all on the video display mode. Its purpose is to indicate to any program that is running whether the video display device that is being used is capable of displaying 80 columns of text. If you are using a normal television set, then it will be nearly impossible to read an 80-column line and so this switch should be put in the down (40) position. However, if you are using a video monitor, then you can put it in the up (80) position.

This method of indicating whether you can support an 80-column screen is not foolproof, however. If the program that is running does not check the switch, then you may well be treated to an unreadable 80-column display on your television set. The only solution is to modify the software so that it takes the setting of the switch into consideration.

The status of the 80/40 switch can be determined by reading a soft switch at RD80SW (\$C060). If the number read is greater than 127, then the switch is in the 40-column position (down); otherwise it is in the 80-column position.

Turning on the Text Display

The IIc uses several input/output (I/O) memory locations as soft switches to control various aspects of the video display as well as several locations that can be read to determine the states of these switches. These locations are summarized in Table 7-1 and we will be referring to them throughout this chapter. Notice that the soft switches are arranged in pairs of locations, one of which turns the switch on and another which turns it off.

To activate a particular soft switch, other than those from \$C050 . . . \$C057, you must write to its location using an Applesoft POKE command or an assembly-language write instruction like STA. You can activate any of the switches from \$C050 . . . \$C057 by either reading or writing. Each pair of on/off soft switches is associated with a status location that can be read to determine the state of the switch. The status is kept in bit 7 which means that the associated switch will be on if the value read from the status location is greater than or equal to 128.

Table 7-1. Video display soft switch and status locations.

Address Hex	(Dec)	Usage	Symbolic Name	Action Taken	Note
\$C000	(49152)	W	80STOREOFF	Allow PAGE2 to switch between video page1 and page2	1
\$C001	(49153)	W	80STOREON	Allow PAGE2 to switch between main and aux. video memory	1
\$C018	(49176)	R7	80STORE	1 = PAGE2 switches main/aux. 0 = PAGE2 switches video pages	1
\$C00C	(49164)	W	80COLOFF	Turn off 80-column display	
\$C00D	(49165)	W	80COLON	Turn on 80-column display	
\$C01F	(49183)	R7	80COL	1 = 80-column display is on 0 = 40-column display is on	
\$C050	(49232)	RW	TEXTOFF	Select graphics mode	
\$C051	(49233)	RW	TEXTON	Select text mode	
\$C01A	(49178)	R7	TEXT	1 = a text mode is active 0 = a graphics mode active	2
\$C052	(49234)	RW	MIXEDOFF	Use full-screen for graphics	2
\$C053	(49235)	RW	MIXEDON	Use graphics with four lines of text	2
\$C01B	(49179)	R7	MIXED	1 = mixed graphics and text 0 = full screen graphics	2
\$C054	(49236)	RW	PAGE2OFF	Select page1 display (or main video memory)	1
\$C055	(49237)	RW	PAGE2ON	Select page2 display (or aux. video memory)	1
\$C01C	(49180)	R7	PAGE2	1 = video page2 selected OR aux. video page selected	1
\$C056	(49238)	RW	HIRESOFF	Select low-resolution graphics	1,2
\$C057	(49239)	RW	HIRESON	Select high-resolution graphics	1,2
\$C01D	(49181)	R7	HIRES	1 = high-resolution graphics 0 = low-resolution graphics	1,2

\$C05E	(49246)	RW	DHIRES0N	Enable double-width graphics	3
\$C05F	(49247)	RW	DHIRES0FF	Disable double-width graphics	3
\$C07F	(49279)	R7	DHIRES	1 = double-width enabled 0 = double-width disabled	3

The "Usage" column in this table indicates how a particular location is to be accessed:

- "W" means "write to the location."
- "RW" means "read from or write to the location."
- "R7" means "read and check bit 7 to determine the status."

Notes:

1. If 80STORE is ON, then PAGE2OFF activates main video RAM (\$400-\$7FF) and PAGE2ON activates auxiliary video RAM. If HIRES is also ON, then PAGE2OFF also activates main high-resolution video RAM (\$2000-\$3FFF) and PAGE2ON also activates auxiliary high-resolution video RAM.
If 80STORE is OFF, then PAGE2OFF turns on text page1 mode and PAGE2 turns on text page2 mode. If HIRES is also ON, then PAGE2OFF also selects high-resolution page1 mode and PAGE2ON selects high-resolution page2 mode.
2. The HIRES and MIXED switches are meaningful only if the TEXT switch is OFF (i.e., a graphics mode is active).
3. The DHIRES switches only take effect if 80COL is ON. DHIRES0N and DHIRES0FF can only be accessed after writing to IOUOFF (\$C07E).

The //c uses the TEXT and 80COL switches to select the video display mode to be used. The TEXT switches are used to select either a graphics mode or a text mode. To select text mode, the TEXTON (\$C051) switch must be accessed (by a read or write operation). This can be done by executing a PEEK(49233) command from Applesoft or a LDA \$C051 command from assembler language. Alternatively, you can use the Applesoft TEXT command.

The //c uses the 80COLOFF (\$C00C) and 80COLON (\$C00D) soft switches to control whether a 40- or 80-column text screen is to be displayed. If you write to 80COLOFF, then the 40-column display will be turned on. To turn on the 80-column display instead, write to 80COLON. A program can always deduce which display mode is currently active by reading the 80COL (\$C01F) status location. If the number read is greater than 127 (that is, bit 7 is on), then the 80-column display is on.

Of course, the PR#3 command that is usually used to enter 80-column mode automatically takes care of properly setting the 80COL switches. Hence, you will usually not have to deal with them directly.

You can see for yourself how the 80COL soft switches work by entering the system monitor so that you can easily access them. Before doing this, make sure that the standard 40-column mode is active by resetting the //c. Then enter CALL -151 from Applesoft and wait for the monitor's "*" prompt to appear. To tell the //c's internal hardware to display an 80-column screen, store any number at 80COLON (\$C00D) by entering the command

```
C00D:0
```

As you will recall from Chapter 3, this command causes a 0 to be stored at \$C00D. (Any other number could also have been stored.) As soon as you do this, the 80-column display will be turned on. Since the special 80-column firmware required to fully support this display mode is not being used, however, the system monitor's video output subroutine will not function properly and only the odd-numbered columns in the display will be used when information is sent to it. To return to a normal 40-column display, enter the command

```
C00C:0
```

to activate the 80COLOFF (\$C00C) switch. Again, any number, not just 0, can be stored at a soft switch location in order to activate the switch.

Text Mode Memory Mapping

There are significant differences in the method the //c uses to display 40-column and 80-column text. We will begin with a discussion of the 40-column text display and then move on to explain how the 80-column text display differs.

40-Column Text Mode

In the 40-column text mode, the screen can be considered to be a matrix of 40 columns by 24 rows. The video subroutines within the system monitor number the rows starting with 0 at the top and ending with 23 at the bottom; the columns are numbered starting with 0 at the left and ending with 39 at the right. Unfortunately, the Applesoft cursor positioning commands, VTAB and HTAB, start numbering the rows and columns with 1. We shall be using the system monitor's numbering system in this section.

In 40-column text mode, the //c translates the contents of one of two 1024-byte blocks of memory (called video RAM) into appropriate images on the video display. The first of these two blocks extends from \$400 . . . \$7FF and is referred to as page1 of text; the other block extends from \$800 . . . \$BFF and is referred to as page2 of text. Note that the word "page" in this context means a block of 1024 bytes of video RAM.

Each character that appears on the 40-column video display screen is defined by one byte in the currently active video page. This means that 64 of the bytes in the 1024-byte block are not used because there are only 960 (40x24) screen locations to be displayed. These unused locations are called "screen holes" and are reserved for use by the firmware that controls the //c's various I/O devices.

To make things as simple as possible, it would be nice if the memory locations used by the video display were mapped linearly to their corresponding coordinates on the video display. If this were the case, then the memory location corresponding to any screen location would be given by $\text{BASE} + (40 \times \text{LINE}) + \text{COLUMN}$, where BASE is the starting address of the video page, LINE is the line number (0 . . . 23), and COLUMN is the column number (0 . . . 39). Unfortunately for all programmers, this is not how the //c handles its mapping of the video display.

Instead, the //c assigns a unique base address to each line on the video screen that is not simply forty positions further into the video memory area from the start of the previous line. The byte at this address and the thirty-nine bytes that immediately follow it in memory are used to represent the forty characters on that video line. Table 7-2 shows the base addresses that are used for the page1 video display and shows how to calculate the address of the byte corresponding to any position on the video display (add 1024 to these addresses to calculate the corresponding page2 addresses). In general terms, if the video line number (0 . . . 23), in binary notation, is given by

`000abcde`

where a . . . e represent bit values, then the 2-byte base address is given by

`0000001cd eabab000`

for page1 addresses or

`000010cd eabab000`

for page2 addresses.

Table 7-2. Text screen video RAM addresses.

<i>Line Number</i>	<i>Base Address</i>	<i>Line Number</i>	<i>Base Address</i>
0	\$400	12	\$628
1	\$480	13	\$6A8
2	\$500	14	\$728
3	\$580	15	\$7A8
4	\$600	16	\$450
5	\$680	17	\$4D0
6	\$700	18	\$550
7	\$780	19	\$5D0
8	\$428	20	\$650
9	\$4A8	21	\$6D0
10	\$528	22	\$750
11	\$5A8	23	\$7D0

- (a) 40-COLUMN SCREEN (columns 0 . . . 39). The address corresponding to a position on the screen is equal to the base address for the line plus the column number.
- (b) 80-COLUMN SCREEN (columns 0 . . . 79). The address corresponding to a position on the screen is equal to the base address for the line plus *one-half* of the column number. If the column number is even, then this address in auxiliary memory is used; if it is odd, then the address in main memory is used.

The base address for the line in which the cursor is currently located is always stored in two zero page locations, called BASL (\$28) and BASH (\$29). To calculate the decimal value of the base address for a given line on the page1 video display from Applesoft, simply move the cursor to the line and then calculate the quantity $\text{PEEK}(40) + 256 * \text{PEEK}(41)$. You can add 1024 to this result to convert it to a page2 base address. Table 7-3 shows a short program that does just this. It positions the cursor with the VTAB command and then calculates the base address using the method just described.

If you want to calculate the base address for a line from assembly language, then use the following instructions:

```
LDA #LINENUM      ;LINENUM=0 . . . 23
JSR BASCALC       ;BASCALC = $FBC1
```

BASCALC is a subroutine within the system monitor (\$FBC1) that does the base address calculation for you. The result will be stored in BASL/BASH (\$28/\$29) and will be equal to the page1 base address. To convert it to the corresponding page2 base address, add \$04 to BASH.

Why does the //c use this strange video mapping scheme? Well, back when the original Apple II was being designed, the main concern was not simplicity of software but rather simplicity of hardware. By changing to this mapping scheme, several chips from the original hardware design could be eliminated, thus making the Apple II less expensive and easier to manufacture. Seven

Table 7-3. BASE.ADDRESSES—a program to display the base addresses for each line on the video screen.

```

0  REM "BASE.ADDRESSES"
50  TEXT : HOME
60  DIM RW(24)
100 FOR I = 1 TO 24
200 VTAB I
300 RW(I) = PEEK (40) + 256 * PEEK
    (41)
400 NEXT I
500 HOME
600 PRINT "THE BASE ADDRESSES F
    OR EACH LINE ARE:"; PRINT
700 FOR I = 1 TO 24 STEP 2
800 PRINT "LINE #"; I - 1; ":"; TAB(
    11); RW(I);
900 PRINT TAB( 20); "LINE #"; I;
    ":"; TAB( 31); RW(I)
1000 NEXT : PRINT

```

years later the new and improved //c was released but, for the sake of compatibility, the video mapping scheme was not changed.

80-Column Text Mode

Since the 80-column screen displays twice as many characters as its 40-column counterpart, another 1024-block of video memory is required to support it. This additional block is not located in the //c's main built-in memory; if this were the case, then the //c would be unacceptably incompatible with its older brothers. Instead, a 1K block of memory that is contained in the auxiliary memory area is used.

This "extra" 1K block actually shares the same addresses used by the main display page, \$400 . . . \$7FF, but, as we have just said, it is in a different physical location. When the //c's 80-column display is active, the video circuitry maps the standard page1 video page locations in main memory to the odd-numbered column positions on the 80-column screen and the auxiliary memory locations to the even-numbered positions. So for any given line on the screen, the contents of columns 0,2,4, . . . ,78 are found in auxiliary memory and the contents of columns 1,3,5, . . . ,79 are found in main memory. The base addresses for each line are the same as for the 40-column screen, however. The mapping scheme used by the 80-column screen is explained in Table 7-2.

You should now be able to see why only odd-numbered columns were used when you experimented with the 80COLON switch earlier. The standard system monitor video output subroutine presumes that a 40-column display is being used and so it accesses the \$400 . . . \$7FF area in main memory only. When 80COL is ON, these locations correspond to odd-numbered columns on the video display; the locations corresponding to even-numbered columns are never accessed by this monitor subroutine.

It is not permissible, of course, to have two physical memory locations, which share the same logical address, active at the same time. The //c uses soft switches to control which of the two \$400 . . . \$7FF areas is to be active so that data can be stored to or read from any 80-column screen position directly. The switches used are PAGE2OFF (\$C054) and PAGE2ON (\$C055). They are used to select the main memory video RAM block and the auxiliary memory video RAM block, respectively, provided that the 80STORE switch is ON. If 80STORE is not ON, then, as we will see below, the PAGE2 switches are used to select between the two different 40-column video pages.

The procedure to follow to store any value to a particular 80-column screen location is as follows:

1. Select the proper mode for the PAGE2 switches by storing any number at 80STOREON (\$C001).
2. Determine the base address for the line required.
3. Divide the required horizontal position (0 . . . 79) by two and add it to the base address.
4. If the horizontal position is odd, then turn on the main \$400 . . . \$7FF video page by storing any number at PAGE2OFF (\$C054). If the position is even, then turn on the auxiliary \$400 . . . \$7FF video page by storing any number at PAGE2ON (\$C055).
5. Store the byte at the address calculated in step 4.
6. Reselect main memory by accessing PAGE2OFF (\$C054).

This is a fairly elaborate procedure but it is handled automatically if you are using the 80-column firmware for video output. It must be followed strictly, however, if you want to POKE data directly into the video screen from your own programs. Table 7-4 shows an Applesoft program called POKE80 that uses this technique to display information on the video screen.

Using Page2 of Text

We have seen how the PAGE2 switches can be used to select between main and auxiliary memory if 80STORE is ON. If 80STORE is OFF, then PAGE2 behaves in quite a different way. That is, it is used to select which of the two available 40-column text pages is to be displayed, the one from \$400 . . . \$7FF (page1) or the one from \$800 . . . \$BFF (page2).

Table 7-4. POKE80—a program to store any data byte in the 80-column video RAM area.

```

0  REM "POKE80"
100 HOME : PRINT CHR$(4);"PR#
    3"
140 INPUT "ENTER LINE # (0...23
    ): ";L
150 INPUT "ENTER COLUMN # (0...
    79): ";C
160 INPUT "ENTER VALUE OF BYTE
    TO BE POKED TO SCREEN (0...2
    55): ";BY
180 VTAB L + 1: REM MOVE CURSOR
    TO PROPER LINE
190 BA = PEEK(40) + 256 * PEEK
    (41): REM GET BASE ADDRESS
200 BA = BA + INT(C / 2): REM
    ADD HORIZ/2
210 IF 2 * INT(C / 2) < > C THEN
    POKE 49236,0: GOTO 230
220 POKE 49237,0: REM SELECT AU
    X MEMORY IF EVEN
230 POKE BA,BY
240 POKE 49236,0: REM SELECT MA
    IN MEMORY
260 VTAB 22: END

```

To select page1, the PAGE2OFF (\$C054) switch must be accessed and to select page2—you guessed it—the PAGE2ON (\$C055) switch must be accessed. You can always tell which page has been selected by reading the PAGE2 (\$C01C) status location. If the number read is greater than 127, page2 is active.

Page1 of the video display is the one that is invariably used by programs, especially if those programs are written in Applesoft. There are two good reasons for this. First, the //c's standard video output subroutines always write screen information to the page1 memory area; if you wanted to send output in the usual way to page2, you would have to write your own subroutines to do this. Second, Applesoft programs are normally stored beginning at location \$801, that is, within page2, which means that your program will be overwritten when the screen display changes. Although it is possible to load an Applesoft program so that it starts beyond page2 at \$C01, this involves using an awkward "preloading" program that looks something like this:

```

100 POKE 103,1:POKE 104,12:POKE 3072,0
200 PRINT CHR$(4);"RUN YOUR.PROGRAM"

```

where YOUR.PROGRAM is the name of the program that you want loaded above page2. Line 100 in the above program stores \$C01 in the Applesoft

beginning-of-program pointer, TXTTAB (\$67), and puts a \$00 byte at \$C00. (A zero byte must always be stored immediately before the start of a tokenized Applesoft program.) See Chapter 4 for a discussion of TXTTAB and other Applesoft pointers.

Page2 does have its uses, however. For example, while page1 is being displayed, a program can be busily writing information on page2 and then, when page2 is complete, the PAGE2ON switch can be accessed to immediately display page2. Then, while page2 is being displayed, page1 can be modified and later switched in by accessing PAGE2OFF. If this process is repeated, extremely good animation effects can be achieved and pages of written information can be displayed very smoothly.

Note that a second 80-column text page (from \$800 . . . \$BFF in main and auxiliary memory) can be selected using PAGE2ON with 80STORE set to OFF and 80COL set to ON. This display mode is not supported by the //c's firmware.

Video Display Attributes: Normal, Inverse, Flash

The //c text screens support three fundamental video display attributes:

- Normal video (white characters on a black background)
- Inverse video (black characters on a white background)
- Flash video (blinking characters)

Every printable ASCII character (that is, those with negative ASCII codes greater than \$9F) can be displayed in normal video without restriction. There are restrictions, however, on what characters can be displayed in inverse and flash video, and these restrictions will depend on which of two possible characters sets available for the //c is currently active.

The two characters sets that the //c supports are called the “primary” character set and the “alternative” character set. When the //c's primary character set is in effect, it is not possible to display flashing or inverse lowercase characters. On the other hand, when the alternative character set is in effect, you will be able to display inverse lowercase characters but you will not be able to display flashing characters. The alternative character set also supports a set of 32 special symbols and icons called MouseText which the primary character set does not. We'll talk more about MouseText later in this section.

One character set or the other can be selected by writing to one of the following two soft switch memory locations:

ALTCHARSETOFF (\$C00E) to select the primary character set

or

ALTCHARSETON (\$C00F) to select the alternative character set

When the //c is in its standard 40-column mode, the default setting of ALTCHARSET is off; when the 80-column firmware is active, the default setting is on. The setting of ALTCHARSET can easily be changed at any time, however, in order to allow either character set to be used in both text modes.

You can determine which character set is currently active by reading the ALTCHARSET status location at \$C01E. If this location is greater than 127 (that is, bit 7 is on), then the alternative set is currently active; otherwise, the primary set is active. The soft switch and status locations that relate to the //c's character sets are summarized in Table 7-5.

Table 7-5. Character set soft switches and status location.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$C00E	(49166)	ALTCHARSETOFF	Select primary character set
\$C00F	(49167)	ALTCHARSETON	Select alternative character set
\$C01E	(49182)	ALTCHARSET	Status of character set switch (> = \$80 if alternative set is active)

The //c examines the two most-significant bits (bits 7 and 6) of each byte that has been stored within the video RAM area in order to determine which attribute is to be used to display the character that it represents. (We will examine the standard monitor subroutines used to store characters in the video RAM area later in this chapter.)

If these two bits are "10" or "11," then the character will be displayed in normal video. If they are "00," the character will be displayed in inverse video. Finally, if they are "01," then the character will be displayed either in flash video, if the primary character set is active, or in inverse video, if the alternative character set is active. These rules are summarized in Table 7-6.

Table 7-6. Video attribute control bits.

<i>Bit 7</i>	<i>Bit 6</i>	<i>Video Attribute</i>
1	1	Normal
1	0	Normal
0	1	Flash (primary character set) Inverse or MouseText (alternative character set)
0	0	Inverse

Table 7-7 shows how the //c interprets each of the 256 possible values that can be stored in its video display memory area, for both the primary and alternative character sets. You can see that the only difference between the two sets is that codes \$40 . . . \$7F represent flashing alphabetic characters and special symbols when the primary set is active, whereas they represent the 32-byte MouseText character set and the inverse lowercase alphabetic and special characters when the alternative set is active.

The program in Table 7-8 will show you visually how the //c's video system interprets each of the 256 possible bytes that might be stored in a video RAM memory location. When you run this program, the name of the currently active character set will be shown at the top of the screen and then eight rows of 32 characters will be displayed, which represent bytes \$00 through \$FF. You can easily select the character set that you want to view by pressing "P" for primary or "A" for alternative after the symbols corresponding to each of the 256 bytes have been displayed. Notice how fast the display changes after you change the character set—this is indicative of a hardware-controlled change rather than a software-controlled change.

MouseText

MouseText symbols and icons are displayed on the video screen whenever a byte between \$40 and \$5F is stored in video RAM memory. As its name suggests, MouseText is primarily used by software that uses the Apple Mouse input device to point to and select commands and functions.

Special subroutines in the //c's 80-column firmware make it fairly simple to display MouseText using standard Applesoft PRINT statements. To do this, you must follow these steps (after the 80-column firmware has been activated and ALTCHAR is ON):

- Turn on inverse video.
- Enable the firmware's handling of MouseText.
- Print the standard keyboard characters that correspond to the special MouseText characters that are to be displayed.
- Turn on normal video.
- Disable the firmware's handling of MouseText.

Here is an example of a short program that does just this:

```
100 PRINT CHR$(4);"PR#3": REM SELECT 80-  
      COLUMN FIRMWARE  
200 PRINT CHR$(27);  
300 PRINT CHR$(15);"@ABCDEFGHIJKLMNOPQRSTUVWXYZ[ ]"  
      ;CHR$(14);  
400 PRINT CHR$(24)
```

Table 7-7. Text screen character display and attributes.

<i>Value of Bytes in Video Page</i>	<i>Symbols Displayed</i>	<i>Display Attribute</i>
\$00-\$1F	@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_	Inverse
\$20-\$3F	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?	Inverse
\$40-\$5F	@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_	Flash (primary)
\$60-\$7F	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?	Flash (primary)
\$40-\$5F	MouseText (see below)	Normal (alternative)
\$60-\$7F	` a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~	Inverse (alternative)
\$80-\$9F	@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_	Normal
\$A0-\$BF	! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?	Normal
\$C0-\$DF	@ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_	Normal
\$E0-\$FF	` a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~	Normal

Here are the MouseText characters and their corresponding ASCII characters:









Table 7-8. CHAR.SETS—a program to display the //c's primary and auxiliary characters sets.

```

0  REM "CHAR.SETS"
100 PRINT CHR$(21): TEXT HOME

110 GOSUB 500
120 FOR I = 0 TO 255
130 HTAB 1 + I - 32 * ( INT (I /
    32))
140 VTAB 3 + I / 32
150 SL = PEEK (40) + 256 * PEEK
    (41) + PEEK (36)
160 POKE SL,I
170 NEXT
180 GOSUB 500
190 VTAB 20: HTAB 1: CALL - 95
    8
200 PRINT "(P)RIMARY OR (A)LTER
    NATIVE? ";: GET A$: PRINT A$

210 IF A$ = "P" OR A$ = "p" THEN
    POKE 49166,0: GOTO 180
220 IF A$ = "A" OR A$ = "a" THEN
    POKE 49167,0: GOTO 180
230 IF A$ = CHR$(27) THEN HOME
    : END
240 GOTO 180
500 VTAB 1: HTAB 1: CALL - 868
    : PRINT "THE ";
510 IF PEEK (49182) > 127 THEN
    PRINT "ALTERNATIVE";: GOTO
    530
520 PRINT "PRIMARY";
530 PRINT " CHARACTER SET IS:"
540 RETURN

```

We're getting a bit ahead of ourselves because this program uses four control characters that won't be discussed until the next section (see Table 7-11). Here's what they mean:

- CHR\$(27)—Tell the firmware to display MouseText characters.
- CHR\$(15)—Turn on inverse video.
- CHR\$(14)—Turn on normal video.
- CHR\$(24)—Turn off the MouseText feature.

When the firmware is told to display MouseText and inverse video is on, the 80-column firmware maps the characters from ASCII code \$C0 ('@') through \$DF ('_') to the MouseText characters having codes from \$40 through \$5F. Thus, in the above program you will not see an inverse video display of keyboard symbols, but rather the complete MouseText character set.

Standard Character Output Subroutines

There is just one standard output subroutine that is used when a program running on the //c wants to send a character to the currently active output device; it is called COUT (\$FDED), which stands for Character OUTput. The Applesoft PRINT command makes use of this subroutine. If the active output device is the video display screen, however, then COUT usually makes use of two other built-in subroutines called COUT1 (\$FDF0), and C3COUT1 (\$C307) to display the character at the proper position on the screen. These subroutines are summarized in Table 7-9.

Table 7-9. Built-in output subroutines

Address		Symbolic Name	Description
Hex	(Dec)		
\$FDED	(65005)	COUT	Sends a character to the currently active output device. The negative ASCII code for the character is in the accumulator.
\$FDF0	(65008)	COUT1	Video output routine used when standard 40-column mode is active.
\$C307	(49927)	C3COUT1	Video output routine used when the 80-column firmware is being used (this includes 80-column mode and the special 40-column mode).

As soon as COUT is called, the following code is executed:

```
JMP (CSWL)
```

which causes the //c to jump to a subroutine that begins at the address stored at CSWL (\$36) and CSWH (\$37). This subroutine is responsible for properly handling the character to be outputted (which is in the accumulator). If the current output device being used is the video display, then this usually means displaying the character on the screen at the current cursor position and advancing the cursor (and scrolling when necessary). If a special control

character is being outputted, then special video control subroutines may be invoked instead. (See below.) Note that by simply changing the address stored at CSWL/CSWH, any output subroutine can be installed on the //c. We will see how to do this later on.

When ProDOS is being used, the address stored at CSWL and CSWH is actually that of a special ProDOS output subroutine. This subroutine will either store information on diskette or display it on the video screen, depending on whether a diskette file is being written to. It also continuously checks to see whether a valid ProDOS command has been printed so that it can execute it immediately. ProDOS commands are easily identified because they are always preceded by a [control-D] character.

If the ProDOS output subroutine needs to display the output on the video screen, then one of two built-in video output subroutines is used. One is called COUT1 (\$FDF0), which is used when in standard 40-column mode. The other is called C3COUT1 (\$C307) and is used when the 80-column firmware is being used. Before calling either of these subroutines, the 65C02 accumulator must be loaded with the ASCII code for the character to be printed (usually with the high bit set to one). If the high bit is zero, the character will be displayed with a special display attribute (either inverse or flashing).

Let's take a closer look right now at exactly what happens when a character is sent to the video display through the //c's output link.

Video Output

As we have seen, COUT will normally pass control to one of two video output subroutines, depending on whether the 80-column firmware is active:

- Control passes to COUT1 (\$FDF0) if the //c is in standard 40-column mode.
- Control passes to C3COUT1 (\$C307) if the 80-column firmware is active.

These subroutines deal with the task of properly displaying a character on the screen or performing special video functions.

After doing some initial housekeeping, both of these subroutines call VIDWAIT (\$FB78). VIDWAIT will, if a carriage return (ASCII code \$8D) is being printed, check the keyboard to see whether a [control-S] has been pressed. If it has, then VIDWAIT pauses until other ASCII code is entered before allowing the character to be printed.

If the character being printed is not a control character (that is, its ASCII code is not between \$80 and \$9F), then control passes to VIDOUT (\$FBFD) if standard 40-column mode is active, or STORCH (\$C3B8) if the 80-column firmware is active. Both of these subroutines store the character in the video RAM page at the currently active cursor position. After the character has been displayed, the cursor is advanced and the subroutine ends. By the way, if

standard 40-column mode is active, the position of the cursor is defined by the values stored at CH (\$24) and CV (\$25), the horizontal and vertical cursor coordinates, respectively. If the 80-column firmware is being used, then the coordinates are kept at OURCH (\$57B) and OURCV (\$5FB) instead.

If the character is a control character, then the subroutines VIDOUT1 (\$FC04) and DOCTL (\$FBF4) will be used to determine whether a special action should be performed. VIDOUT1 is used when standard 40-column mode is active and it reacts in a special way to four control characters only: [control-G] (bell), [control-H] (backspace), [control-J] (line feed), and [control-M] (carriage return). The actions that are taken when any of these control characters is encountered are shown in Table 7-10. All other control characters are ignored by VIDOUT1.

Table 7-10. Special control codes used by both COUT1 and C3COUT1.

<i>Control Code</i>	<i>Description</i>
[control-G] \$87	Bell. Beep the speaker
[control-H] \$88	Backspace. Move the cursor one position to the left or to the end of the previous line if already at left edge.
[control-J] \$8A	Line feed. Move the cursor down one line.
[control-M] \$8D	Carriage return. Initiates a carriage return/line feed sequence that moves the cursor to the left position of the next line.

DOCTL is used when the 80-column firmware is active. The first thing it does is to call the VIDOUT1 subroutine in order to handle a [control-G], [control-H], [control-J], or [control-M] character. If the character was one of these four, then DOCTL ends. If not, then tests are made for the presence of several other control characters which are listed in Table 7-11. If one of these special control characters is found, then the special function associated with it will generally be executed. It is possible, however, to disable these functions by entering an ESC [control-D] escape sequence from the keyboard before printing the control character. This sequence was discussed in Chapter 6, as was the ESC [control-E] sequence that is used to re-enable the special control characters.

After a control character is handled by VIDOUT1 or DOCTL, the video output subroutine ends and control is passed back to the calling subroutine.

Table 7-11. Special control codes used by C3COUT1 (80-column firmware only).

Control Code	Description
[control-K] \$8B	<i>Clear to end of screen.</i> Clear from the current cursor position to the end of the screen.
[control-L] \$8C	<i>Form feed.</i> Clear the screen and move the cursor to the home position (top left-hand corner).
[control-N] \$8E	<i>Normal.</i> Turn on normal video display.
[control-O] \$8F	<i>Inverse.</i> Turn on inverse video display.
[control-Q] \$91	<i>40-column.</i> Keep 80-column firmware active, but move to a 40-column display.
[control-R] \$92	<i>80-column.</i> Move to an 80-column display.
[control-U] \$95	<i>80-off.</i> Turn off the 80-column firmware and return to 40-column format.
[control-V] \$96	<i>Scroll down.</i> Scroll the display down one line leaving the cursor where it is.
[control-W] \$97	<i>Scroll up.</i> Scroll the display up one line leaving the cursor where it is.
[control-X] \$98	<i>Mouse characters off.</i> Disable the displaying of the special mouse character set.
[control-Y] \$99	<i>Home.</i> Move the cursor to the home position.
[control-Z] \$9A	<i>Clear line.</i> Clear the entire line on which the cursor is positioned.
[control-[] \$9B	<i>Mouse characters on.</i> Enable the displaying of the special mouse character set.
[control-\] \$9C	<i>Forward.</i> Move the cursor forward one space with wraparound.
[control-]] \$9D	<i>Clear to end of line.</i> Clear the screen from the current cursor position to the end of the line.
[control-_] \$9F	<i>Move cursor up one line (in the same column).</i> If the cursor is already at the top, it will not move.

Video Screen Windowing

When the //c is first turned on, the standard output subroutines will automatically use the entire video screen for text display. It is possible to define a smaller “window,” however, into which all output is to be confined. The

advantage of defining such a window is that information outside the window will not usually be overwritten. When it becomes necessary to perform a scrolling operation, only the contents of the window will be moved; the information outside of the window will stay put.

The dimensions of the text window can be set by adjusting four locations in zero page, described in Table 7-12. These locations are used to set the leftmost column position of the window (WNDLFT), the first line number used by the window (WNDTOP), the bottom line number used by the window plus one (WNCBTM), and the width, in characters, of the window (WNCWDTH).

You can change the window parameters with simple Applesoft POKE statements. If you do change them, however, keep in mind the following two rules:

- WNCBTM must always be greater than WNCBTM.
- WNCWDTH + WNDLFT must not exceed the maximum display width (40 or 80).

Table 7-12. Text window parameters.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$20	(32)	WNDLFT	Left side of window (40 col.: 0 . . . 39) (80 col.: 0 . . . 79)
\$21	(33)	WNCWDTH	Width of window (40 col.: 1 . . . 40) (80 col.: 1 . . . 80)
\$22	(34)	WNCBTM	Top of window (0 . . . 23)
\$23	(35)	WNCBTM	Bottom of window + 1 (1 . . . 24)

After the window parameters have been changed, you can quickly and easily restore them to their initial default values by entering the Applesoft TEXT command.

How COUT1 and C3COUT1 Set the Video Attribute

As we have seen, a character is normally displayed on the video screen by loading the 65C02 accumulator with its ASCII code (with the high bit on) and then calling COUT (\$FDED). COUT, in turn, calls either COUT1 (\$FDF0), if the standard 40-column mode is active, or C3COUT1 (\$C307), if the 80-column firmware is being used. These subroutines take care of displaying the character at the proper position on the screen.

How, then, does the `//c` determine whether to display the character in normal, inverse, or flash video? The answer depends on whether COUT1 or C3COUT1 is being used.

If COUT1 is being used, then just before a printable ASCII code (that is, everything above \$9F) is sent on to the main part of the video output routine, it is logically ANDed with the number stored at INVFLG (\$32). The purpose of doing this is to adjust bits 6 and 7 of the outgoing character code so that they are equal to the values needed to select the required video attribute (see Table 7-6). The value stored at INVFLG is called a “mask” because it will hide (clear to 0) those bits in the character code that are 0 in the INVFLG byte, but will leave unaffected those bits that are 1 in the INVFLG byte. The three values for INVFLG, which are used to select the normal, flash, and inverse attributes, respectively, are set out in Table 7-13.

Table 7-13. INVFLG mask values

<i>Value of INVFLG</i>	<i>Video Attribute</i>	<i>Effect on Character Code</i>
\$FF	Normal video	No effect
\$7F	Flash video	Clears bit 7
\$3F	Inverse video	Clears bits 7 and 6

INVFLG is location \$32.

Note: With \$7F in INVFLAG, characters with values from \$A0 . . . \$BF will not flash, they will be displayed in inverse video (see text).

With some exceptions, if you take any printable ASCII code and logically AND it with each of the three values for INVFLG, you will see that the bits will be set in accordance with the rules set out in Table 7-6. The exceptions relate to the use of the flash mask (\$7F) with those characters having ASCII codes from \$A0 . . . \$BF. Bit 6 of these codes is 0, so when bit 7 is cleared to 0 by the flash mask, the attribute bits for an inverse character are set up and so the character won't flash. In an assembly-language program, you can circumvent this problem by always logically ORing a character code with the value \$40 to force bit 6 to 1 before calling COUT1. This is exactly what is done by the Applesoft interpreter when you are running an Applesoft program.

If C3COUT1 is being used, then INVFLG is still examined before storing the character code in the video RAM area, but only bit 7 is checked. If it is one, the character will be displayed in normal video; if it is zero, it will be displayed in inverse video. Although it is possible to display a flashing character on the 80-column screen, the 80-column firmware does not support this attribute.

The Applesoft NORMAL, FLASH, and INVERSE commands are all used to select the value stored at INVFLG. Keep in mind, however, that if the 80-column firmware is being used, the FLASH command will only cause an

inverse video display; if you want to display flashing characters, you will have to POKE bytes directly to the video RAM area.

Changing Output Devices: The OUTPUT Link

Character output on the //c is usually sent to built-in system monitor sub-routines that control the //c's 40-column or 80-column video display screens. It is possible, however, to send output to other peripheral devices that are connected to the //c through its built-in interface ports. Examples of such devices are a disk drive, a printer, and a modem.

The //c uses the same general method to handle output to such devices that it uses to handle input. This method was discussed in detail in Chapter 6 in the section describing the //c's input link.

We mentioned earlier that the first instruction in the standard COUT character output routine looks like this:

```
JMP ($0036)
```

As we explained when discussing the input link, this is called an "indirect jump" instruction and it will cause the //c to transfer control to the address stored at location \$36 (low byte) and location \$37 (high byte). If you are using the standard 40-column output routine, \$36/\$37 will contain the address of a subroutine in ProDOS that in turn usually calls COUT1 (\$FDF0). (It could also call another subroutine to write information to a diskette file instead.) By changing the address stored at \$36/\$37, you can redirect the //c to any other output subroutine that you care to execute, including one used by an alternative output device.

The symbolic name for locations \$36/\$37 is CSW (for character switch); \$36 by itself is called CSWL and \$37 is called CSWH. CSW is commonly referred to as the "output link" or "output hook."

You will recall from Chapter 6 that the Applesoft "IN#s" command can be used to redirect input to port "s". In a similar way, you can use "PR#s" to redirect output to port "s". When "PR#s" is entered, a program beginning at location \$Cs00 (where s is the port number), which is the first location in a ROM area dedicated to that port, is executed. Typically, the program beginning at this location will modify CSW so that it will point to a new output routine also contained in ROM. Note that if a PR#0 command is entered, then the address of COUT1 (\$FDF0), the //c's standard 40-column output subroutine, will be stored at CSW.

Subject to complications that arise whenever ProDOS is being used (see below), you can also change the output link directly by using the Applesoft POKE command or the assembler's STA command to store the address of the new input routine directly into CSW at \$36 and \$37. This address can be in either ROM or RAM.

Designing a CSW Output Subroutine

Any CSW output subroutine that will be used to replace the standard ones used by the //c must adhere to certain rules relating to the usage of 65C02 registers. First of all, the output subroutine must examine the accumulator to determine which character code is being passed to it. Second, the subroutine must end with the A, X, and Y registers unaffected. If it is necessary to change the contents of these registers in the body of the subroutine, the registers must first be saved and then restored just before the subroutine ends.

Replacing the Video Output Subroutine

One common reason for changing the CSW output subroutine is simply to modify to the manner in which character output to the video display is handled. For example, you may want to perform one of the following tasks:

- Redefine the effect of control characters on the video display or define special actions to be performed by previously unused control characters.
- Prevent certain characters from being displayed.
- Translate character codes from one encoding system to another.

For relatively minor changes such as these, it is not necessary to rewrite all the underlying code that takes care of positioning the cursor and displaying characters on the video display. What can be done instead is to install a new output subroutine that performs its special chores and then, if necessary, passes control to the standard output subroutine that can then handle the relatively complex chores of displaying a character on the screen and executing special video-control commands.

Here is an example of a short input subroutine that preprocesses character output before passing it on (if necessary) to the standard output subroutine:

```
NEWOUT      CMP  #$87           ;Is this a bell?
             BNE NOCHANGE      ;No, so branch
             RTS               ;Yes, so do nothing
NOCHANGE    JMP  COUT1         ;Perform normal output
```

This subroutine will prevent a bell character from ever being sent to the standard output subroutine (meaning that you won't hear that annoying beep when you make an error). It works by continually comparing each character code that is printed with the ASCII "bell" code (code \$87) and by simply executing an RTS instruction if one is found. If the character code is not a bell, control passes directly to the standard video output subroutine.

ProDOS and the Output Link

The same restrictions referred to in Chapter 6 that apply when changing the input link when ProDOS is active also apply when changing the output

link. When ProDOS is first activated, the address stored in CSW is copied to an internal ProDOS output link location and then the address of a special ProDOS output subroutine is placed in CSW. This subroutine is responsible for detecting and handling any ProDOS commands that are printed (they are preceded by a [control-D] character) and for writing information to a diskette file if a ProDOS WRITE or APPEND command is in effect. If ProDOS is not currently writing to a file, then it will send output to the subroutine whose address is stored in the ProDOS output link. This is initially one of the standard video output subroutines.

Normal attempts to store new addresses directly to CSW will obviously lead to a disconnection of ProDOS. Rather than repeat the explanations given in Chapter 6, we shall simply state how the output link must be changed to ensure that both ProDOS and the new output subroutine will be active. Any one of the following procedures may be used:

- Use the PR# command while in Applesoft direct mode (not within a program) or use the command

```
PRINT CHR$(4);"PR#s"
```

from within a program (where "s" represents the port number).

- Use the BRUN command to load and execute an assembly-language program that stores the new output address into CSW.
- Use the POKE command to store the new input address directly into the ProDOS output link locations at \$BE30 and \$BE31. Alternatively, use the Applesoft CALL command or the system monitor GO command to execute an assembly-language program that stores the address directly into \$BE30 and \$BE31.

If you are using ProDOS, you can also use a special form of the PR# command to properly install an output subroutine that is located anywhere in memory and not just in the ROM area for a port. The output subroutine must, however, begin with a 65C02 "CLD" (clear decimal) instruction. To install the output subroutine, execute a statement of the form

```
PRINT CHR$(4);"PR# Aaddr"
```

from within an Applesoft program, where "addr" represents either the decimal starting address of the new output subroutine or, if preceded by "\$", the hexadecimal starting address.

Low-Resolution Graphics Mode

The //c also supports two general graphic display modes called low-resolution graphics and high-resolution graphics. These modes are primarily used to present non-text information such as pictures, graphs, and maps and will now be described in detail, beginning with low-resolution graphics mode.

Turning on the Low-Resolution Graphics Display

The easiest way to activate the //c's low-resolution graphics display is to enter the Applesoft GR command from Applesoft direct mode. This command, however, selects only one of four possible versions of low-resolution graphics (namely, page1 with mixed graphics/text). As we will see later, other versions must be activated by directly setting some of the //c's video soft switches.

When the standard low-resolution graphics mode is in effect, colored "blocks" are displayed on the screen instead of text symbols. The dimensions of the screen are 40 blocks wide by 48 blocks deep (or 40 blocks deep if a special mixed mode is in effect—see below). Column positions range from 0 on the left to 39 on the right; row positions range from 0 on the top to 47 on the bottom.

There are two possible pages of low-resolution graphics that can be displayed on the //c. The video RAM area that defines the first display screen (page1) extends from \$400 . . . \$7FF, and the area that defines the second (page2) extends from \$800 . . . \$BFF. These are the same video RAM areas used to support the two pages of text mode.

To turn on either page of standard low-resolution graphics, you must first ensure that the PAGE2 switches (PAGE2OFF and PAGE2ON) can be used to select which of the two graphics pages is to be used rather than to select whether main memory or auxiliary memory is to be used. This can be done by writing to 80STOREOFF (\$C000). (To be safe, you might also want to write to IOUDISON (\$C07E) and DHIRESOFF (\$C05F) to ensure that double-width low-resolution graphics are not accidentally enabled. As we shall see in the next section, these commands will cause the circuitry that enables this special graphics mode to be disabled.)

To turn on page1 of low-resolution graphics, the following switches must be "thrown" by reading from or writing to all of the following soft switch memory locations:

TEXTOFF (\$C050)—selects a graphics mode
HIRESOFF (\$C056)—selects low-resolution graphics
PAGE2OFF (\$C054)—selects page1

To turn on page2, throw the following switches by reading from or writing to all of the following locations:

TEXTOFF (\$C050)
HIRESOFF (\$C056)
PAGE2ON (\$C055) —selects page2

In addition, it will be necessary to throw one of two other switches that control whether full screen graphics will be displayed or whether four lines of text will be "mixed in" at the bottom of the screen with 40 lines of low-resolution graphics above them. The switches that control this are MIXEDON

(\$C053), which enables mixed graphics and text, and MIXEDOFF (\$C052), which enables full-screen graphics. Simply read from or write to these memory locations to activate these switches.

The switches that must be accessed to turn on the four different combinations of low-resolution graphics display modes are summarized in Table 7-14.

Table 7-14. Low-resolution graphics display modes.

<i>Page1 of Low-Resolution Graphics (full-screen mode)</i>	<i>Page2 of Low-Resolution Graphics (full-screen mode)</i>
TEXTOFF (\$C050)	TEXTOFF (\$C050)
HIRESOFF (\$C056)	HIRESOFF (\$C056)
MIXEDOFF (\$C052)	MIXEDOFF (\$C052)
PAGE2OFF (\$C054)	PAGE2ON (\$C055)
<i>Page1 of Low-Resolution Graphics (mixed mode)</i>	<i>Page2 of Low-Resolution Graphics (mixed mode)</i>
TEXTOFF (\$C050)	TEXTOFF (\$C050)
HIRESOFF (\$C056)	HIRESOFF (\$C056)
MIXEDON (\$C053)	MIXEDON (\$C053)
PAGE2OFF (\$C054)	PAGE2ON (\$C055)

Low-Resolution Graphics Screen Memory Mapping

Each block on the low-resolution graphics screen is defined by one-half of a byte (four bits) that is stored within the currently active video RAM area (\$400 . . . \$7FF for page1 or \$800 . . . \$BFF for page2). The number stored in this half byte is the color code for the block (see the next section). Table 7-15 shows the mapping scheme for each block on page1 of the low-resolution graphics screen; page2 addresses can be calculated by adding 1024 to the corresponding addresses for page1. Note that the base addresses for each pair of lines in the graphics screen (that is, 0/1, 2/3, 4/5, . . . ,46/47 are the same as those for text lines 0, 1, 2, . . . ,23.)

Low-Resolution Graphics Colors

A special color code is stored in 4 bits of the byte in the video RAM page that corresponds to a particular block position. As Table 7-15 indicates, these 4 bits are found in the top half of the byte (bits 4 . . . 7) or the bottom half (bits 0 . . . 3), depending on the block's position on the screen. Table 7-16 contains a list of the color codes that can be stored in the byte in video RAM in order to generate the sixteen different colors that the low-resolution graphics mode supports.

Table 7-15. Low-resolution graphics video RAM screen addresses.

<i>Line Number</i>	<i>Base Address</i>	<i>Line Number</i>	<i>Base Address</i>
0,1	\$400	24,25	\$628
2,3	\$480	26,27	\$6A8
4,5	\$500	28,29	\$728
6,7	\$580	30,31	\$7A8
8,9	\$600	32,33	\$450
10,11	\$680	34,35	\$4D0
12,13	\$700	36,37	\$550
14,15	\$780	38,39	\$5D0
16,17	\$428	49,41	\$650
18,19	\$4A8	42,43	\$6D0
20,21	\$528	44,45	\$750
22,23	\$5A8	46,47	\$7D0

- (a) STANDARD LOW-RESOLUTION GRAPHICS (columns 0 . . . 39). The address corresponding to a position on the screen is equal to the base address for the line plus the column number. If the line number is even, then the lower 4 bits of the byte stored at this address are used to store the color code; if it is odd, the upper 4 bits are used.
- (b) DOUBLE-WIDTH LOW-RESOLUTION GRAPHICS (columns 0 . . . 79). The address corresponding to a position on the screen is equal to the base address for the line plus *one-half* of the column number. If the column number is even, then this address in auxiliary memory is used; if it is odd, the address in main memory is used. If the line number is even, then the lower 4 bits of the byte stored at this address are used to store the color code; if it is odd, the upper 4 bits are used.

Double-Width Low-Resolution Graphics

The //c also supports a special double-width low-resolution graphics mode that is not available on the earlier Apple II and Apple II Plus models. It is available as an option on the Apple //e but the big difference is that the Applesoft low-resolution graphics commands have been modified on the //c so that they will support this new mode. With the //e you have to write your own graphics commands before you can efficiently use this mode.

Unlike standard low-resolution graphics, only one page of double-width graphics is available. Just as for 80-column text mode, the PAGE2 switches normally used to flip between display pages are instead used to select whether the part of the double-width graphics video page within main memory or auxiliary memory is to be used.

Turning on Double-Width Low-Resolution Graphics

The double-width low-resolution graphics can be displayed by first setting the TEXTOFF (\$C050) soft switch to select a graphics mode, HIRESOFF

Table 7-16. Low-resolution graphics color codes.

<i>Color Code</i>	<i>Color</i>
\$00	Black
\$01	Magenta
\$02	Dark blue
\$03	Purple
\$04	Dark green
\$05	Gray1
\$06	Medium blue
\$07	Light blue
\$08	Brown
\$09	Orange
\$0A	Gray2
\$0B	Pink
\$0C	Light green
\$0D	Yellow
\$0E	Aquamarine
\$0F	White

Note: these codes relate to bytes in main memory only (see Table 7-17 for the corresponding codes for bytes in auxiliary memory when using double-width low-resolution graphics).

(\$C056) to select low-resolution graphics, and either MIXEDOFF (\$C052) to select full-screen graphics or MIXEDON (\$C053) to select 40 lines of graphics with 4 lines of text. This can be done by executing the following assembly-language instructions:

```
STA $C050
STA $C056
STA $C052 (or STA $C053)
```

If you do this while you are in standard 40-column mode, the normal-width low-resolution graphics screen will be displayed. To enable the double-width graphics, three further soft switches must be set: 80COLON (\$C00D), IOUDISON (\$C07E), and DHIRESON (\$C05E). As we saw when discussing 80-column text mode, the 80COLON switch is used to turn on the double-width display mode. The IOUDISON switch simply enables access to the DHIRESON switch. (As we will see in Chapter 10, the I/O locations used by the DHIRESON switches are used by some mouse switches as well; the IOUDISON switch is used to select which switches are to be active.) The DHIRESON switch allows you to turn on the double-width graphics support circuitry. (It can be turned off by writing to IOUDISON (\$C07E) and DHIRESOFF (\$C05F).) To perform these steps from an assembly-language program, you would use the following three instructions:

```

STA $C00D
STA $C07E
STA $C05E

```

An easier way to turn on the double low-resolution graphics screen is to use standard Applesoft commands. To throw the same series of switches that we have just outlined from Applesoft, you would use this program segment:

```

100 PRINT CHR$(4);"PR#3": REM This sets 80COLON
200 POKE 49278,0: REM Enable access to DHIRES switch
300 POKE 49246,0: REM Access DHIRESON
400 GR : REM This sets low-res graphics switches

```

Once you have turned on double-width low-resolution graphics in this way, you can use the Applesoft graphics commands to draw on the screen. For example, to put a white border around the screen, execute the following program lines:

```

500 COLOR= 15: REM Select white blocks
600 HLIN 0,79 AT 0: VLIN 0,39 AT 79: REM Top, Right
700 HLIN 0,79 AT 39: VLIN 0,39 AT 0: REM Bottom, Left

```

Double-Width Low-Resolution Graphics Screen Memory Mapping

The //c displays double-width low-resolution graphics in much the same way that it displays its 80-column text screen. That is, the region of memory from \$400 . . . \$7FF that resides on the 80-column text card (auxiliary memory) is interleaved with the same region of memory on the motherboard (main memory). For a given low-resolution graphics screen line, all even locations are mapped to locations in auxiliary memory and all odd locations are mapped to locations in main memory. This mapping scheme is described in Table 7-15.

You can select which area of screen memory is to be accessed by first ensuring that the 80STORE switch is on (by writing to location \$C001). This allows the PAGE2 switches to be used to select between main and auxiliary memory rather than page1 and page2 of graphics. PAGE2OFF (\$C054) is used to select main memory and PAGE2ON (\$C055) is used to select auxiliary memory. As you can see, writing to the double-width low-resolution graphics screen is done in exactly the same way as writing to the 80-column text screen. After accessing auxiliary memory in this way, you should always turn off PAGE2 by accessing PAGE2OFF (\$C054).

Note that there is a second page of double-width low-resolution graphics that occupies \$800 . . . \$BFF in main and auxiliary memory. It can be selected by setting 80STOREOFF, 80COLON, and PAGE2ON

Double-Width Low-Resolution Graphics Colors

Because of timing differences in interpreting auxiliary memory, the color codes stored in auxiliary memory to set the color of the low-resolution graph-

ics blocks are different from the standard ones set out in Table 7-16. These new color codes are set out in Table 7-17 in the standard color order of Table 7-16.

Table 7-17. Low-resolution graphics color codes for auxiliary memory locations.

<i>Color Code</i>	<i>Color</i>
\$00	Black
\$08	Magenta
\$01	Dark blue
\$09	Purple
\$02	Dark green
\$0A	Gray1
\$03	Medium blue
\$0B	Light blue
\$04	Brown
\$0C	Orange
\$05	Gray2
\$0D	Pink
\$06	Light green
\$0E	Yellow
\$07	Aquamarine
\$0F	White

Built-In Support for Low-Resolution Graphics

The easiest way to manipulate the standard low-resolution graphics screen is to use the Applesoft commands designed for this purpose. These commands are briefly summarized in Table 7-18. These commands will work with both the single- and double-width low-resolution graphics modes.

Table 7-18. Applesoft low-resolution graphics commands.

<i>Command</i>	<i>Description</i>
GR	Turns on page1 of low-resolution graphics in mixed mode and clears the display.
COLOR =	Selects a low-resolution color number.
PLOT	Plots a block on the screen.
HLIN	Draws a horizontal line on the screen.
VLIN	Draws a vertical line on the screen.
SCRN	Gets the color code at a given screen position.

Support for low-resolution graphics is also afforded by a series of subroutines contained within the //c's system monitor. These subroutines are described in Table 7-20 and the zero page locations that they use are set out in Table 7-19. Note that some zero page locations must be properly set up before calling these subroutines. In particular, COLOR (\$30) must contain the desired 4-bit color code (in both halves of the byte), H2 (\$2C) must contain the destination location of a horizontal line before HLINE (\$F819) is called, and V2 (\$2D) must contain the destination location of a vertical line before VLINE (\$F828) is called. Note, however, that these subroutines do not work properly when the double-width low-resolution graphics mode is enabled. Refer to Rob Moore's article entitled "80-Column //e Low-Res Graphics" (noted in the references at the end of this chapter) for examples of assembly-language subroutines that do support the double-width mode.

Table 7-19. Zero page locations used by low-resolution graphics subroutines.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$26	(38)	GBASL	Low byte of graphics screen line base address.
\$27	(39)	GBASH	High byte of graphics screen line base address.
\$2C	(44)	H2	Horizontal destination location for drawing a horizontal line.
\$2D	(45)	V2	Vertical destination location for drawing a vertical line.
\$2E	(46)	MASK	Contains \$F0 or \$0F and is used to clear out the proper 4-bit area before setting the color for a low-resolution block.
\$30	(48)	COLOR	Contains the color code for the low-resolution block in the upper 4 bits and the lower 4 bits.

High-Resolution Graphics Mode

Of the two main graphics modes that the //c supports, high-resolution graphics mode is probably the most useful and exciting. This is because, as the name of this mode suggests, the points that can be plotted on the screen (called "pixels", for picture elements) are much smaller than low-resolution graphics blocks, thus allowing you to draw much finer shapes. This allows

Table 7-20. System monitor low-resolution graphics subroutines.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$F800	(63488)	PLOT	Plot a block using the current color at the position given by A (vertical) and Y (horizontal).
\$F819	(63513)	HLINE	Draw a horizontal line beginning from the position given by A (vertical) and Y (horizontal). The ending horizontal position is stored at H2 (\$2C).
\$F828	(63528)	VLINE	Draw a vertical line beginning from the position given by A (vertical) and Y (horizontal). The ending vertical position is stored at V2 (\$2D).
\$F832	(63588)	CLRSCR	Clear the full low-resolution graphics screen to black.
\$F836	(63542)	CLRTOP	Clear the top 40 lines of the low-resolution graphics screen to black.
\$F847	(63559)	GBASCALC	Put the base address for the line number contained in the accumulator (0 . . . 47) into GBASL (\$26) and GBASH (\$27).
\$F864	(63558)	SETCOL	Set up the color mask at location COLOR (\$30). On entry, A contains the color code (0 . . . 15).
\$F871	(63601)	SCRN	Determine the color code stored at the location given by A (vertical) and Y (horizontal).

you not only to place easily recognizable images on the screen but also to place more images on the screen. No wonder that virtually all popular games now being released for the //c use high-resolution graphics.

Turning on the High-Resolution Graphics Display

The //c supports two pages of high-resolution graphics, each of which are defined by a block of 8192 bytes. Page1 of high-resolution graphics is mapped to the area from \$2000 . . . \$3FFF and page2 is mapped to \$4000 . . . \$5FFF.

The dimensions of the full-size high-resolution screens are 280 pixels wide by 192 pixels high. A mixed mode can also be defined, however, where the bottom 32 lines of pixels are replaced by 4 lines of text so that the dimensions of the graphics screens become 280x160. Numbering of both the pixel rows and the pixel columns begin at 0 and the (0,0) position is at the top left-hand corner of the screen.

Each pixel on the display screen is controlled by one bit of a byte in the 8K area associated with that screen and can be made to appear as one of eight colors, with some restrictions. If that bit is off, then a black dot will be displayed on the screen; if it is on, one of the five other colors (white, green, violet, orange, or blue) will be displayed. The two other colors are a duplicate white and black. We'll take a closer look at how to generate colored images later in this chapter.

You can quickly turn on the two high-resolution screens from Applesoft by using the HGR and HGR2 commands. HGR turns on mixed-mode page1 high-resolution graphics, and HGR2 turns on full-screen page2 high-resolution graphics. Let's take a closer look at how the //c's video soft switches can be used directly to select the various high-resolution graphics display modes.

To turn on either page of standard high-resolution graphics, you must first ensure that the PAGE2 switches (PAGE2OFF and PAGE2ON) can be used to select which of the two graphics pages is to be used rather than to select whether main memory or auxiliary memory is to be used. This can be done by writing to 80STOREOFF (\$C000). (To ensure that double-width high-resolution graphics are not accidentally enabled, you should also write to IOUDISON (\$C07E) and DHIRESOFF (\$C05F). As we shall see in the next section on double-width high-resolution graphics, this will disable the circuitry that enables this special graphics mode.)

The high-resolution graphics displays are turned on in much the same way as the low-resolution displays. In fact, the only difference is that the HIRESON (\$C057) soft switch must be accessed instead of the HIRESOFF (\$C056) soft switch. To turn on page1, read from or write to the following locations (with 80STORE in the off position):

TEXTOFF (\$C050)—selects a graphics mode
HIRESON (\$C057)—selects high-resolution graphics
PAGE2OFF (\$C054)—selects page1

To turn on page2, simply access PAGE2ON (\$C055) instead of PAGE2OFF.

You can also control whether full screen graphics are to be displayed or whether four lines of text are to appear at the bottom of the screen instead of the last 32 lines of the graphics page. The switches to use to control these two options are MIXEDON (\$C053), which selects the graphics-text combination, and MIXEDOFF (\$C052), which selects full-screen graphics.

Table 7-21 summarizes the switches that must be set to select each of the four possible combinations of high-resolution display modes.

Table 7-21. High-resolution graphics display modes.

<i>Page1 of High-Resolution Graphics (full-screen mode)</i>	<i>Page2 of High-Resolution Graphics (full-screen mode)</i>
TEXTOFF (\$C050)	TEXTOFF (\$C050)
HIRESOON (\$C057)	HIRESOON (\$C057)
MIXEDOFF (\$C052)	MIXEDOFF (\$C052)
PAGE2OFF (\$C054)	PAGE2ON (\$C055)
<i>Page1 of High-Resolution Graphics (mixed mode)</i>	<i>Page2 of High-Resolution Graphics (mixed mode)</i>
TEXTOFF (\$C050)	TEXTOFF (\$C050)
HIRESOON (\$C057)	HIRESOON (\$C057)
MIXEDON (\$C053)	MIXEDON (\$C053)
PAGE2OFF (\$C054)	PAGE2ON (\$C055)

High-Resolution Graphics Screen Memory Mapping

The `//c` uses 40 consecutive bytes in the applicable high-resolution screen video RAM memory area (\$2000 . . . \$3FFF, page1, or \$4000 . . . \$5FFF, page2) to define the contents of each 280-pixel graphics line. The most-significant bit of each of these bytes, however, is not used for display purposes (it is used to select which of two sets of four colors can be displayed). Each of the $40 \times 7 = 280$ active bits in these 40 bytes corresponds to a unique column position. The seven pixels corresponding to each byte in memory are displayed on the screen in reverse order of their positions within the byte. That is, the first pixel displayed on the screen (the one farthest to the left) corresponds to bit 0, the next one corresponds to bit 1, and so on. If a bit is set to "1", then the pixel will be illuminated; if it is cleared to "0", it will be turned off.

As with the text screen, the high-resolution page1 and page2 memory areas are not mapped linearly to the video screen. To determine the memory address corresponding to a particular pixel, it is first necessary to calculate the base address for the line in which it appears. Reverting to binary notation for a moment, if the line number (0 . . . 191) is given by

abcde fgh

(where a . . . h represent values of bits 7 . . . 0, respectively), then the base address for that line is given by the two bytes

0ppfghcd eabab000

where

pp = 01 for page1
pp = 10 for page2

The base addresses for each line of the high-resolution display are set out in Table 7-22. To convert these addresses to the corresponding page2 addresses, add \$2000 (8192).

Table 7-22. High-resolution graphics video RAM screen addresses.

<i>Line Number</i>	<i>Base Address</i>	<i>Line Number</i>	<i>Base Address</i>
0–7	\$2000 + \$400xRLN	96–103	\$2228 + \$400xRLN
8–15	\$2080 + \$400xRLN	104–111	\$23A8 + \$400xRLN
16–23	\$2100 + \$400xRLN	112–119	\$2328 + \$400xRLN
24–31	\$2180 + \$400xRLN	120–127	\$23A8 + \$400xRLN
32–39	\$2200 + \$400xRLN	128–135	\$2050 + \$400xRLN
40–47	\$2280 + \$400xRLN	136–143	\$20D0 + \$400xRLN
48–55	\$2300 + \$400xRLN	144–151	\$2150 + \$400xRLN
56–63	\$2380 + \$400xRLN	152–159	\$21D0 + \$400xRLN
64–71	\$2028 + \$400xRLN	160–167	\$2250 + \$400xRLN
72–79	\$20A8 + \$400xRLN	168–175	\$22D0 + \$400xRLN
80–87	\$2128 + \$400xRLN	176–183	\$2350 + \$400xRLN
88–95	\$22A8 + \$400xRLN	184–191	\$23D0 + \$400xRLN

RLN = relative line number. This number is equal to the actual line number minus the first line number in the group of eight within which it falls in the above table. For example, RLN for line #83 is 3 (83–80).

- (a) STANDARD HIGH-RESOLUTION GRAPHICS (columns 0 . . . 279). The address of the byte corresponding to a pixel position is equal to the base address for the line plus the horizontal pixel position divided by 7. The bit position within this byte corresponding to the pixel is the horizontal pixel position modulo 7.
- (b) DOUBLE-WIDTH HIGH-RESOLUTION GRAPHICS (columns 0 . . . 559). The address of the byte corresponding to a pixel position is equal to the base address for the line plus the horizontal pixel position divided by 14. If the horizontal pixel position modulo 14 is between 0–6, this address in auxiliary memory is used; if it is between 7–13, this address in main memory is used. The bit position within the byte corresponding to the pixel is the horizontal pixel position modulo 7.

The byte position number (0 . . . 39) for a particular pixel column (remember that 7 columns are defined by one byte) is given by the quotient of

$$X / 7$$

where X is the column number (0 . . . 279). To access this byte, the 65C02 indirect-indexed addressing mode, “(zp),Y”, can be used (“zp” refers to any zero page location that contains the low half of the base address for the line; zp + 1 contains the high half). The bit number within this byte that is mapped to the column is given by the remainder generated by the $X/7$ calculation (that is, X modulo 7). This is the bit that can be set to 1 to illuminate a pixel on the screen or cleared to 0 to turn it off.

High-Resolution Graphics Colors

Pixels on the high-resolution graphics screen can be one of eight colors: black1, black2, white1, white2, green, orange, violet, and blue. These are the eight colors that can be set using the Applesoft HCOLOR = command. Because of the way the high-resolution graphics circuitry works on the //c, however, you cannot display all colors at all positions on the high-resolution screen. For example, green and orange pixels can appear only in odd-numbered columns, and violet and blue pixels can appear only in even-numbered columns. In addition, in some circumstances that we will refer to in a moment, you cannot display blue and orange pixels close to green and violet pixels, and vice versa.

If you are plotting points in a particular color, you must ensure that, even if a particular column is selected, you do not illuminate pixels in that column if it is a restricted column for that color, or else the color will be wrong. This is handled automatically by the Applesoft high-resolution graphics commands and can be done from assembly language by logically ANDing the byte that is to be stored in the video page with the appropriate color mask. This mask will ensure that no "1"s can appear in restricted columns. Table 7-23 sets out the column restrictions and color masks for each of the eight allowed high-resolution graphics colors.

Table 7-23. High-resolution screen display information.

Color	Applesoft HCOLOR =	Value of High-Order Bit of Display Byte	Display Byte Mask		Column Restr.
			Even Byte	Odd Byte	
Black1	0	0	\$00	\$00	None
Green	1	0	\$2A	\$55	Odd only
Violet	2	0	\$55	\$2A	Even only
White1	3	0	\$7F	\$7F	None
Black2	4	1	\$80	\$80	None
Orange	5	1	\$AA	\$D5	Odd only
Blue	6	1	\$D5	\$AA	Even only
White2	7	1	\$FF	\$FF	None

Not all colors can be used at the same time. The most-significant bit of the byte that defines the pixel must be cleared to 0 in order to have a '1' in the byte displayed as green/violet or set to 1 to have it displayed as orange/blue (for an odd/even column). A side effect of this phenomenon is that it is not possible to generate green and violet pixels if they are defined by bits in the same byte as orange and blue pixels.

To get white displayed on the screen, two horizontally adjacent pixels on the screen must be set to 1. If this is done, then both pixels will be displayed as white. Note that there are two different types of white, white1 and white2. The only difference between these two colors is the status of the high-order bit within the byte that defines the two adjacent pixels. Note, also, that it is not possible to get a single white dot surrounded by black because an isolated '1' bit will be interpreted as either green/violet or orange/blue.

In summary, the standard high-resolution screen looks at each horizontally adjacent pair of bits to determine which of four colors is to be displayed: black1 (00), white1 (11), green (01), or violet (10), if bit 7 in the byte in which they are contained is off; or black2 (00), white2 (11), orange (01), or blue (10), if bit 7 is on.

It should now be clear that because of the column restrictions on colors other than black and white, the effective screen resolution is only 140x192 for color graphics even though it is possible to control the states of all 280 horizontal pixels individually.

Animation with High-Resolution Graphics

One of the main reasons for including two high-resolution graphics pages on the //c was to allow you to generate high-quality animation effects. Animation is typically simulated on a computer by first drawing a shape, pausing, erasing the original shape, and then redrawing it at its new position. By repeating this procedure, the effect of motion is created.

If this procedure is used in connection with one display screen only, then the problem of "flickering" can arise and the first shape will not appear to change smoothly into the next. This effect is observed because the screen is continually being "redrawn" by the electronic circuitry within the video display unit before the first shape has been completely erased and redrawn. If the shape is complex enough, a partially erased or partially redrawn shape will be displayed for discernible periods of time.

One way of getting around this problem is to draw the next shape in an animation sequence on the graphics page that is not being displayed and then, after it has been so drawn, to throw the switch that activates that page of graphics. Then, while that page is being displayed, the shape on the other page can be erased and repositioned, and then that page can be displayed again. The net effect is that all erasing and redrawing is done on the screen that is not being displayed and so flickering will be eliminated. If Applesoft graphics commands are being used, the page that is being written to can be controlled simply by adjusting the value of the byte located at \$E6. To write to page1, this byte must be set equal to \$20; to write to page2, it must be set equal to \$40.

One problem with using the two pages of high-resolution graphics in this way, however, is that another 8K of memory must be devoted for use by the

display screen and is unavailable for use by the program. For larger programs, this can be a major limitation indeed.

Fortunately, there is an alternative method that can be used to achieve flicker-free animation: moving a shape while the video display unit is not actually refreshing the screen. This method can be used on the //c and //e only and not on the earlier Apple II and Apple II Plus models.

The video display unit is continually “refreshing” the screen by redrawing all the scan lines that define the display screen. It does this by moving an electron beam in a zig-zag motion across the display screen from top to bottom. After all of the video scan lines have regenerated in this way, there is a synchronization delay during which the electron beam is repositioned to the upper left-hand corner of the screen awaiting the arrival of the next video frame. This delay occurs every 1/60 of a second.

The delay between the end of one zig-zag scan and the beginning of the next one is called the vertical blanking interval, and during this time the screen display is not being altered in any way. Thus, if during this vertical blanking interval we could change the data bytes that define the screen display image in such a way as to cause the shape being animated to be erased and repositioned, there would be no discernible flickering.

Well, we can! It is possible, under software control, to enable a special vertical blanking (VBL) interrupt signal, that will interrupt the 65C02 at the beginning of every vertical retrace operation (that is, 60 times per second). If your interrupt- handling subroutine is properly set up, you can use it to erase and redraw your animated shape before the retracing operation ends (it lasts for about 12,000 machine cycles) so that there will be no flickering.

We’ll defer a complete discussion of how to activate the VBL interrupt to Chapter 10 where we’ll also discuss how it is used in connection with the Apple Mouse.

Double-Width High-Resolution Graphics

The //c also supports an impressive double-width high-resolution graphics display mode. When it is used, the other “half” of the double-width graphics screen is stored in auxiliary memory. Unlike the double-width low-resolution graphics mode, however, neither Applesoft nor the system monitor contains any commands or subroutines that allow you to use this mode directly. Programs are available, however, that will allow you to take advantage of the power of this graphics mode; some of them are listed in the references at the end of this chapter.

The double-width high-resolution graphics mode has a pixel resolution of 560x192, rather than the standard 280x192, and allows a total of sixteen colors! These colors are the same ones that can be displayed when using standard low-resolution graphics.

Turning on Double-Width High-Resolution Graphics

It is relatively simple to activate the double-width high-resolution graphics mode. The first step is to turn on page1 of high-resolution graphics mode as you would normally. This can be done by executing the following sequence of instructions:

```
STA $C050—TEXTOFF (enables graphics)
STA $C057—HIRESO (high-resolution)
STA $C053—MIXEDON (mixed graphics/text)
```

The next step is to enable the double-width mode by setting the 80COLON (\$C00D), IOUDISON (\$C07E), and DHIRESON (\$C05E) switches to enable the double-width graphics circuitry. You can set these switches by executing these three instructions:

```
STA $C00D—80COLON (sets double-width switch)
STA $C07E—IOUDISON (enables access to DHIRESON switch)
STA $C05E—DHIRESON (enables double-width graphics)
```

You can also turn on the same series of switches from Applesoft by running the following program:

```
100 PRINT CHR$(4);"PR#3": REM THIS SETS 80COLON
200 POKE 49278,0: REM ENABLE ACCESS TO DHIRESON SWITCH
300 POKE 49246,0: REM ACCESS DHIRESON
400 HGR : REM THIS SETS HIGH-RES GRAPHICS SWITCHES
```

Once the double-width graphics screen has been activated, the next step is to draw something on it. This is easier said than done, however, because the Applesoft high-resolution graphics commands work only with the standard 280-column screen. If you attempt to use them, you will see rather strange effects, since only the screen area in main memory will be used. For example, try entering the Applesoft commands

```
HCOLOR=3
HPLOT 0,0 TO 279,0
```

If you were to do this for normal-width high-resolution graphics you would see a horizontal white line drawn across the top of the screen. With double-width graphics enabled, however, the white line is "broken" at forty different positions. The data bytes for these positions are contained in auxiliary memory and are not dealt with by Applesoft.

See the references at the end of this chapter for sources of programs that support double-width high-resolution graphics.

Double-Width High-Resolution Graphics Screen Memory Mapping

You will recall that when the //c is displaying double-width text (that is, 80 columns of text) or double-width low-resolution graphics, it interleaves the

video RAM bytes in main memory with those contained at the same addresses in auxiliary memory. Well, double-width high-resolution graphics works in exactly the same way. The region of memory from \$2000 . . . \$3FFF in main memory is interleaved with an 8K block of memory having the same addresses on the extended 80-column text card in such a way that of the 80 consecutive bytes used to define the contents of one line (recall that only 40 were required for standard high-resolution graphics), the even ones (0, 2, 4, . . . , 78) are found in auxiliary memory and the odd ones in main memory. The mapping scheme used is summarized in Table 7-22.

Just as in standard high-resolution graphics mode, each of the 80 bytes corresponds to seven consecutive pixels on the screen. The first pixel is controlled by bit 0, the next one by bit 1, and so on. Bit 7 is not used.

The 80STORE switch enables you to select which of the two \$2000 . . . \$3FFF blocks you want to read from or write to. By setting 80STOREON (by writing to location \$C001), the PAGE2 switches can be used to select either the 8K block in main memory, by accessing PAGE2OFF (\$C054), or the 8K block in auxiliary memory, by accessing PAGE2ON (\$C055). After you have written to screen memory, you should always access PAGE2OFF (\$C054) to re-enable main video memory.

Note that there is a second page of double-width high-resolution graphics that occupies \$4000 . . . \$5FFF in main and auxiliary memory. It can be selected by setting 80STOREOFF, 80COLON, and PAGE2ON.

Double-Width High-Resolution Graphics Colors

When we discussed normal high-resolution graphics, we saw how the //c interprets two adjacent pixels as one of four colors. Not surprisingly, when double-width graphics are used, the //c interprets four adjacent pixels as one of sixteen different colors ($2^4 = 16$). The 4-bit pixel patterns that give rise to these colors are set out in Table 7-24. Since pixels are displayed on the video screen in the reverse order that they appear in the video RAM data bytes, these patterns must be reversed to obtain the corresponding bit patterns that must be stored in memory to generate them.

Note that the high bit of each of the 80 bytes that is used to store information for each line of double-width graphics is not used at all—not even to affect the colors generated by the bits within that byte (as it is in normal high-resolution graphics).

Built-In Support for High-Resolution Graphics

Applesoft contains several commands that are used to control various aspects of the two standard high-resolution graphics screens. These commands are summarized in Table 7-25.

The //c's system monitor does not support high-resolution graphics at all. The Applesoft ROM does, however, contain several built-in subroutines that can be used from an assembly-language program in order to draw points,

Table 7-24. Bit patterns for the sixteen double-width high-resolution graphics colors.

<i>Color</i>	<i>Bit Pattern</i>
Black	0000
Dark red	1000
Dark blue	0100
Purple	1100
Dark green	0010
Gray1	1010
Medium blue	0110
Light blue	1110
Brown	0001
Orange	1001
Gray2	0101
Pink	1101
Green	0011
Yellow	1011
Light green	0111
White	1111

Table 7-25. Applesoft high-resolution graphics commands.

<i>Command</i>	<i>Description</i>
HGR	Turns on page1 of high-resolution graphics in mixed mode and clears the screen.
HGR2	Turns on page2 of high-resolution graphics in full-screen mode and clears the screen.
HCOLOR =	Selects the high-resolution color number.
HPLOT	Plots pixels and draws lines on the screen.
DRAW	Draws a shape on the screen in the color set by HCOLOR = .
XDRAW	Draws a shape on the screen using the complement of the color already existing at each plotted point.
ROT =	Sets the rotation factor used when drawing shapes.
SCALE =	Sets the scale factor used when drawing shapes.

lines, and shapes. These subroutines are set out in Table 7-27 and the zero page locations that they use are set out in Table 7-26.

Note that these commands and subroutines do not support double-width high-resolution graphics at all.

Table 7-26. Zero page locations used by the Applesoft high-resolution graphics subroutines.

Address		Symbolic Name	Description
Hex	(Dec)		
\$E0	(224)	HHORIZ (low)	Horizontal coordinate (0 . . . 279).
\$E1	(225)		
\$E2	(226)	HVERT	Vertical coordinate (0 . . . 191).
\$E4	(228)	HMASK	High-resolution color mask.
\$E6	(230)	HPAG	High-resolution page designation. Set this byte to \$20 for page1 and to \$40 for page2.
\$E7	(231)	SCALE	Applesoft SCALE= factor for shapes.
\$F9	(249)	ROT	Applesoft ROT= factor for shapes.

Further Reading for Chapter 7

Standard reference works . . .

80-Column Text Card Manual, Apple Computer, Inc., 1982.

Extended 80-Column Text Card Supplement, Apple Computer, Inc., 1982.

Note: the above two reference manuals are written for the optional 80-column text card used on the Apple //e only. However, much of the information in them is directly applicable to the built-in 80-column display on the //c as well.

On changing the output link . . .

G. Little, "Paged Printer Output for the Apple", *Micro*, October 1980, pp. 47–48. This article demonstrates how to change the output link so that the format of printed output can be controlled.

On ProDOS and the output link . . .

C. Fretwell, "Setting I/O Hooks in ProDOS", *Call -A.P.P.L.E.*, April 1984, p. 39

On high-resolution graphics . . .

B. Bishop, "Apple II Hires Picture Compression", *Micro*, November 1979, p. 17.

- L. Spurlock, "Understanding Hi-Res Graphics", *Call -A.P.P.L.E.*, January 1980, p. 6. An analysis of the high-resolution mapping scheme.
- B. Bishop, "Apple II Hi-Res Graphics: Resolving the Resolution Myth", *Apple Orchard*, Fall 1980, pp. 7–10. Discussion of the mapping of the high-resolution graphics screen.
- E.C. So, "Picture Compression", *Call -A.P.P.L.E.*, May 1982, p. 21.
- R.T. Simoni, Jr., "A New Shape Subroutine for the Apple", *Byte*, August 1983, pp. 292–309. A new method for drawing high-resolution shapes that leads to flicker-free animation.

On double-width graphics . . .

- R. Moore, "80-Column //e Low-Res Graphics", *Call -A.P.P.L.E.*, July 1983, pp. 9–13. A set of subroutines supporting double-width low-resolution graphics is presented in this article.
- D. Worth, "Hi-Res Double Play", *Softalk*, July 1983, pp. 120–126. A description of the Apple //e's double-width high-resolution graphics. It is applicable to the //c as well.
- P. Baum and L. Roddenberry, "Applesoft Brushes for Double Hi-Res Art", *Softalk*, September 1983, pp. 82–99. Programs are presented that support double-width high-resolution graphics.
- A. Watson III, "True Sixteen-Color Hi-Res", *Apple Orchard*, January 1984, pp. 26–46. An excellent discussion of the theory of double-width high-resolution graphics. A set of assembly-language driver programs are also presented which can be called from Applesoft.
- Extended 80-Column Text Card Supplement, Apple Computer, Inc., 1982.
- R.R. Devine, "Double Hi-Res Graphics I", *Nibble*, May 1984, pp. 81–96. Another detailed discussion of the double-width display mode.
- R.R. Devine, "Double Hi-Res Graphics II", *Nibble*, August 1984, pp. 122–129.
- R.R. Devine, "Double Hi-Res Graphics III", *Nibble*, September 1984, pp. 139–144.

On video display theory . . .

- J. Hockenhull, "Video Interfacing", *Call -A.P.P.L.E.*, June 1982, pp. 9–13. A good discussion of the theory of video display technology.
- J. Mazur, "Hardtalk", *Softalk*, April 1983, pp. 215–225. A technical analysis of the Apple II video display system.
- J. Mazur, "Hardtalk", *Softalk*, May 1983, pp. 91–98. A technical analysis of the Apple II video display system.
- R.H. Sturges, Jr., "Double the Apple II's Color Choices", *Byte*, November 1983, pp. 449–463. A good explanation of how the Apple II generates colored images.

Table 7-27. Applesoft ROM high-resolution graphics subroutines.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$F3D8	(62424)	HGR2	Turns on high-resolution page2 (full-screen) and clears it to black.
\$F3E2	(62434)	HGR	Turns on high-resolution page1 (with 4 lines of text) and clears it to black.
\$F457	(62551)	HPLOT	Plots a colored dot at the position given by A (vertical), Y (horizontal high) and X (horizontal low).
\$F53A	(62778)	HLIN	Draws a line from the last plotted dot to the position given by Y (vertical), X (horizontal high), and A (horizontal low).
\$F601	(62977)	DRAW	Draws the shape whose data area is pointed to by Y (high) and X (low) using the rotation factor in A. The shape is drawn by inverting the existing screen bits that are used by the shape.
\$F65D	(63069)	XDRAW	Same as DRAW except that when the shape is plotted, the existing screen bits and the shape bits are logically exclusive-ORed with each other to determine the new value of the screen bit.
\$F6EC	(63212)	SETHCOL	Sets the active code to the value of X (0 . . . 7). These are the eight colors defined by the Applesoft HCOLOR = command.

8

Memory Management

As we saw in Chapter 2, the 65C02 microprocessor that controls the //c is capable of addressing only 65536 (64K) different logical memory locations. These locations have addresses that range from \$0000 to \$FFFF. A standard //c, however, contains many more physical memory locations than this.

A detailed memory map of the //c was presented at the end of Chapter 2. In summary, the memory that is built-in to the //c is as follows:

- 64K of main RAM memory
- 10K of ROM memory for Applesoft
- 2K of ROM memory for the standard system monitor
- 0.25K of I/O memory
- 3.75K of ROM memory that contains extensions to the standard system monitor and support for built-in peripheral devices (two serial ports, mouse port, disk drive, and 80-column display)
- 64K of auxiliary RAM memory (1K of which is used by the 80-column display circuitry)

If you add up all the numbers, the total comes to 144K. This may seem a bit surprising since we just said that the //c's 65C02 microprocessor is capable of addressing only 64K locations. How is all that extra memory used? To answer this, you must realize that the 65C02 can use as much memory as you care to provide to it so long as there are never more than 64K physical memory locations active at the same time and so long as no two active memory locations are associated with the same address. Several soft switches are available on the //c that allow you to easily select which one of those duplicated memory areas is to be active. The technique used to select memory in this way is called "bank-switching."

In this chapter, we will be looking at the soft switches that the //c uses to control usage of its duplicated memory areas, and we will show how they can be used to take advantage of all of the memory available on the //c.

16K Bank-Switched RAM Areas

The //c comes with 64K of main RAM memory which is normally used by Applesoft and ProDOS. This memory, however, is not mapped to one contiguous area of memory encompassing the entire 64K space that the 65C02 is capable of addressing. The first 48K of this memory space corresponds to the contiguous block of memory from \$0000 . . . \$BFFF but the remaining 16K of memory, which is called "bank-switched RAM," corresponds to one 8K region of memory from \$E000 . . . \$FFFF and two 4K regions of memory from \$D000 . . . \$DFFF.

The addresses used by bank-switched RAM are exactly the same as those used by the Applesoft ROM and the standard system monitor ROM. A memory map of the alternative main memory areas from \$D000 . . . \$FFFF is shown in Figure 8-1.

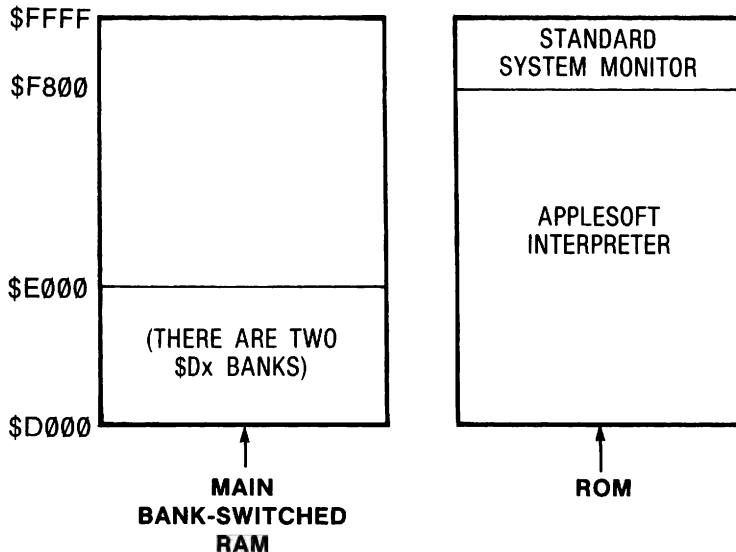


Figure 8-1. Alternative main memory areas from \$D000 . . . \$FFFF.

The 16K bank-switched RAM on the //c traces its roots to the earlier Apple II or Apple II Plus models. On those models, the 16K of bank-switched RAM was introduced to the system by inserting a special 16K memory expansion card into slot 0 of those systems. The original reason for adding this memory was to provide needed space for the extremely large Apple Pascal Operating System. The extra memory, however, can also be used for conventional data and program storage. In fact, ProDOS occupies much of bank-switched RAM.

The //c reserves several I/O memory locations for use as soft switches to control whether bank-switched RAM or the corresponding ROM space is to be active. As we will see in the following section, we can even set these switches

in such a way that the RAM area will be active for write operations at the same time that the corresponding ROM area is active for read operations, or so that the RAM area can be read from but not written to.

Using Bank-Switched RAM

As we have seen, the 16K of bank-switched main RAM in the //c is made up of one 8K area that is mapped to the addresses \$E000 . . . \$FFFF and two different 4K areas that are mapped to the addresses \$D000 . . . \$DFFF. These 4K areas are commonly referred to as “banks.”

Unfortunately, there are two schools of thought on how to refer to these two 4K memory banks: sometimes they are referred to as banks 0 and 1 and sometimes as banks 1 and 2. For our purposes, we will use the latter nomenclature.

The sixteen I/O addresses in the range \$C080 . . . \$C08F are used as soft switches to control the operation of the bank-switched RAM. Switches are available to select which of the two 4K banks is to be used, to enable the bank-switched RAM for reading, for writing, or for both reading and writing. Note that the bank-switched RAM does not have to be enabled for reading and writing at the same time. This means that you can be writing to the RAM area while running a program that uses subroutines in the ROM that occupies the same memory locations (that is, subroutines in Applesoft and the system monitor).

To activate the particular mode of operation that is desired, it is necessary to select the appropriate soft switch address and then perform any kind of read operation at that address, for example, an LDA, LDY, LDX, or BIT instruction in assembly language or a PEEK from Applesoft.

The addresses that are to be used to control the operation of bank-switched RAM are of the form \$C08X, where X represents the four least-significant bits of the address. Figure 8-2 indicates the general function of each of these bits; only three of these bits are used.

I/O Address: \$C08X

X =

BANK— SELECT	UNUSED	READ— SELECT	WRITE— SELECT
bit 3	bit 2	bit 1	bit 0

1=bank 1

0=bank 2

1	1	→ read RAM/write RAM
0	1	→ read ROM/write RAM
1	0	→ read ROM/write ROM
0	0	→ read RAM/write ROM

Figure 8-2. Bank-switched RAM control bits.

The functions of each of the three active bits are as follows:

Bank-Select Bit (bit 3). This bit indicates which of the two \$D000-\$DFFF memory banks is to be used. If the bit is set to 1, then bank 1 will be selected; if it is cleared to 0, then bank 2 will be selected.

Read-Select Bit (bit 1). This bit, in conjunction with the write-select bit, indicates the read status of bank-switched RAM. If the bit is set equal to the value of the write-select bit, then locations in bank-switched RAM will be read from when an address in the range \$D000 . . . \$FFFF is specified; otherwise, the corresponding locations in ROM will be used.

Write-Select Bit (bit 0). This bit indicates the write status of bank-switched RAM. If the bit is 1, *and the I/O address is read twice in succession*, then locations in bank-switched RAM will be written to when an address in the range \$D000 . . . \$FFFF is specified; otherwise, the corresponding locations in ROM will be used.

There are eight different ways of turning on and off these three control bits, and each of the eight different addresses generated controls bank-switched RAM in an unique way. The function of each of the eight unique bank-switched RAM soft switches is summarized in Table 8-1.

Table 8-1. Bank-switched RAM soft switches.

<i>Hex</i>	<i>Address (Dec)</i>	<i>Symbolic Name</i>	<i>Active \$Dx Bank</i>	<i>Read From</i>	<i>Write to RAM?</i>
\$C080	(49280)	READBSR2	2	RAM	No
\$C081	(49281)	WRITEBSR2	2	ROM	Yes*
\$C082	(49282)	OFFBSR2	2	ROM	No
\$C083	(49283)	RDWRBSR2	2	RAM	Yes*
\$C088	(49288)	READBSR1	1	RAM	No
\$C089	(49289)	WRITEBSR1	1	ROM	Yes*
\$C08A	(49290)	OFFBSR1	1	ROM	No
\$C08B	(49291)	RDWRBSR1	1	RAM	Yes*

A location must be read from to perform the indicated function.

*Read twice in succession to write-enable bank-switched RAM.

Reading the Status of Bank-Switched RAM Soft Switches

Any program that changes the soft switches that control the state of bank-switched RAM should properly restore them to their original states when the program ends. (If it doesn't, the next program executed may not perform properly.) This can easily be done on the //c because there are two I/O status locations, called RDBANK2 (\$C011) and RDLCRAM (\$C012), that can be read

to determine the current state of the bank-switched RAM switches. These two locations are summarized in Table 8-2.

Table 8-2. Bank-switched RAM status locations.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$C011	(49169)	RDBANK2	If this location is $\geq \$80$, then Bank2 of bank-switched RAM has been selected; if not, Bank1 has been selected.
\$C012	(49170)	RDLGRAM	If this location is $\geq \$80$, then bank-switched RAM has been read-enabled; if not, the corresponding ROM locations are enabled.

A program that saves the two bank-switched RAM status values and then uses them to restore the original state of bank-switched RAM would look something like this:

```

        LDA RDBANK2          ;Save bank status
        STA BANKSAVE
        LDA RDLGRAM          ;Save read-enable status
        STA READSAVE
        .
        [the program fiddles with
         bank-switched RAM here]
        .
        LDA BANKSAVE          ;Get bank status
        BPL SETBANK1          ;Branch if bank1 selected
        LDA READSAVE          ;Get read-enable status
        BPL SETROM            ;Branch if ROM selected
        LDA $C083              ;Read RAM, bank2
        LDA $C083              ; (write-enable)
        RTS
SETROM   LDA $C081              ;Read ROM, bank2
        LDA $C081              ; (write-enable)
        RTS
SETBANK1 LDA READSAVE          ;Get read-enable status
        BPL SETROM1           ;Branch if ROM selected
        LDA $C08B              ;Read RAM, bank1
        LDA $C08B              ; (write-enable)
        RTS
SETROM1  LDA $C089              ;Read ROM, bank1
        LDA $C089              ; (write-enable)
        RTS

```

Since there is no status location available for determining the write-enable status of bank-switched RAM, you always have to “guess” what it was. The best guess is that it was write-enabled because even if your guess is wrong, no program should be trying to write to bank-switched RAM without first write-enabling it anyway. In keeping with this, those soft switches that write-enable bank-switched RAM were used in the above example (remember that they must be read twice in succession).

Auxiliary Bank-Switched RAM

There is another 16K bank-switched RAM area available in the //c’s 64K auxiliary memory space.

The same soft switches that are used to control the main bank-switched RAM area are used to control the bank-switched RAM area in auxiliary memory. Before you can read to or write from this part of auxiliary memory, however, you will also have to use another set of switches that control, among other things, which of the two bank-switched RAM areas is to be used. These switches are ALTZPOFF (\$C008) and ALTZPON (\$C009) and are described in Table 8-3. The status of the switch is held in ALTZP (\$C016).

Table 8-3. Auxiliary bank-switched RAM soft switches.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(Dec)</i>		
\$C008	(49160)	ALTZPOFF	Enable main bank-switched RAM + main zero page/stack
\$C009	(49161)	ALTZPON	Enable auxiliary bank-switched RAM + auxiliary zero page/stack
\$C016	(49174)	ALTZP	Status: > = \$80 is ON, < \$80 is OFF

The ALTZP switches are used not only to select which of the two bank-switched RAM areas is to be used, but also to select which of two 65C02 zero pages (\$00 . . . \$FF) and stacks (\$100 . . . \$1FF) are to be used. As you might expect, the //c keeps its “spare” zero page and stack in auxiliary memory and the “original” ones in main memory. This means that as soon as the ALTZPON switch is set, the main zero page and stack are disengaged and unless the program that is running realizes this and adjusts for it, it might just end up in the twilight zone.

To avoid such problems, the program must always set ALTZPOFF as soon as it is finished dealing with auxiliary bank-switched RAM but after it has returned from all subroutines that it has called since it first set ALTZPON. The return addresses for these subroutines are stored in the auxiliary stack

and not the main stack and will be lost when the main stack is restored. For similar reasons, the program must never return from a subroutine that was called before ALTZPON was set until ALTZPOFF is restored. Furthermore, before setting ALTZPON, the program should move to a safe part of memory all zero page locations that it will be using while ALTZP is ON. Once ALTZP is ON, it can move them into the same locations in the auxiliary zero page. It should repeat this process when going in the other direction (that is, from ALTZPON to ALTZPOFF) so that no zero page information is lost.

Playing with Bank-Switched RAM

If you want to store information (programs or data) in bank-switched RAM, then you must first write-enable the portion of bank-switched RAM that you want to write to, store the information at the desired locations in the \$D000 . . . \$FFFF address space, and then write-protect bank-switched RAM. The programming sequence to use to do this would be as follows:

```
LDA $C081          ;Two accesses will write-
                   ;enable
LDA $C081          ;Bank-switched RAM (bank 2)
[store information]
LDA $C082          ;Write-protect and set
                   ;ROM read
```

To read information (programs or data) contained in bank-switched RAM, or to execute programs that reside there, you must first enable bank-switched RAM for reading, read the information or execute the program, and then re-enable reading of the ROMs. The programming sequence would be as follows:

```
LDA $C080          ;read-enable bank-switched
                   ;RAM (bank 2)
[read information]
LDA $C082          ;re-enable ROM read
```

The latter method can be used to execute assembly-language programs only. The reason that Applesoft programs cannot be made to execute while residing in bank-switched RAM is that the place where the program is stored and the Applesoft ROM area must be active at the same time and this just isn't possible because bank-switched RAM and the Applesoft interpreter use the same addresses.

Note that if you are running assembly-language programs that reside in bank-switched RAM, you must make absolutely sure that those programs do not use subroutines contained in the Applesoft or standard system monitor ROMs. The reason is simple: as far as the //c is concerned, as soon as you read-enable bank-switched RAM, the //c doesn't think the ROMs exist and so the system will "hang" when it attempts to execute a ROM subroutine. If you really must use these ROM subroutines, you must first execute a JSR instruction to a location in normal RAM that contains code that first deselects bank-

switched RAM for reading and selects the ROMs (\$C082), calls the ROM subroutine, and then read-enables bank-switched RAM (\$C080) and executes an RTS instruction to return to the program in bank-switched RAM.

To avoid these software complexities, you could move the ROM code that you need to use into bank-switched RAM by write-enabling bank-switched RAM and then performing a memory move from the ROMs to the same memory locations in bank-switched RAM. When this is done, the program can call the “pseudo-ROM” locations directly.

You should bear in mind one more important consideration when using bank-switched RAM. Do not attempt to deselect bank-switched RAM for reading while running a program that is contained in bank-switched RAM. If you try to do this, the motherboard ROMs will immediately be enabled and your program, which is still executing at the same address in RAM, will suddenly enter limbo because its code has been “replaced” by the internal ROM code. Any deselection of bank-switched RAM must be done by a program segment that resides in “normal” RAM (from \$0000 . . . \$BFFF).

Bank-Switched RAM and ProDOS

If you are using ProDOS then you should not try to use the main bank-switched RAM area for data or program storage. The reason for this is simple: ProDOS uses this area of memory to hold its operating system subroutines. If you overwrite this area, you will almost certainly crash the system. Use the bank-switched RAM area in auxiliary memory instead.

Auxiliary RAM Memory Area

“Auxiliary” memory is an extra 64K memory space which is built into the //c and is mapped to addresses in exactly the same way as main RAM memory. As we saw in Chapter 7, 1K of auxiliary memory from \$400 . . . \$7FF is used to support the //c’s 80-column display mode and another 8K from \$2000 . . . \$3FFF is used to support the double-width high-resolution graphics display mode. In addition, auxiliary memory is used as a special RAMdisk volume called /RAM when ProDOS is active. In the following sections, we will be describing in detail how to use auxiliary memory.

There are several soft switches that are used to control auxiliary memory. We have already discussed some of these in Chapter 7, when we looked at how to control the 80-column text display and double-width graphics displays. In addition, in the previous section, we saw that the upper 16K of auxiliary memory is functionally identical to main memory’s bank-switched RAM and can be selected or deselected by making use of the ALTZPON and ALTZPOFF switches.

In this section, we will examine all the other soft switches that control auxiliary memory and elaborate further on the ones that have previously been discussed.

Using Auxiliary Memory

There are three main groups of switches that control the status of auxiliary memory. These are the ALTZP switches (“ALTeRnate ZeRo Page”), the RAMRD (“RAM ReaD”) switches, and the RAMWRT (“RAM WRiTe”) switches; they are summarized in Table 8-4.

Table 8-4. Auxiliary memory soft switch and status locations.

Address		Symbolic Name	Description
Hex	(Dec)		
\$C002	(49154)	RAMRDOFF	Read main memory from \$200-\$BFFF
\$C003	(49155)	RAMRDON	Read aux. memory from \$200-\$BFFF
\$C013	(49171)	RAMRD	Status: > = \$80 is ON, < \$80 is OFF
\$C004	(49156)	RAMWRTOFF	Write main memory from \$200-\$BFFF
\$C005	(49157)	RAMWRTON	Write aux. memory from \$200-\$BFFF
\$C014	(49172)	RAMWRT	Status: > = \$80 is ON, < \$80 is OFF
\$C008	(49160)	ALTZPOFF	Select main memory from \$0-\$1FF and enable main 16K bank from \$D000-\$FFFF
\$C009	(49161)	ALTZPON	Select aux. memory from \$0-\$1FF and enable aux. 16K bank from \$D000-\$FFFF
\$C016	(49174)	ALTZP	Status: > = \$80 is ON, < \$80 is OFF

The ALTZP Switch

We briefly discussed ALTZP earlier in this chapter when we looked at the bank-switched RAM contained in auxiliary memory. The ALTZP switches control two blocks of memory that are duplicated in main and auxiliary memory. First, they are used to select whether the 65C02 zero page and stack areas (\$0000 . . . \$01FF) in main memory or in auxiliary memory are to be used. Second, they are used to select whether main bank-switched RAM or auxiliary bank-switched RAM is to be used.

The ALTZPON (\$C009) switch is used to select auxiliary memory and the ALTZPOFF (\$C008) switch is used to select main memory. The current status of this switch can be determined by reading ALTZP (\$C016); if the value read from this location is greater than 127, then ALTZP is ON; otherwise it is OFF. Note that you must write to the ALTZPON and ALTZPOFF switches in order to use them. Figure 8-3 indicates which memory areas are switching whenever the ALTZP switches are written to.

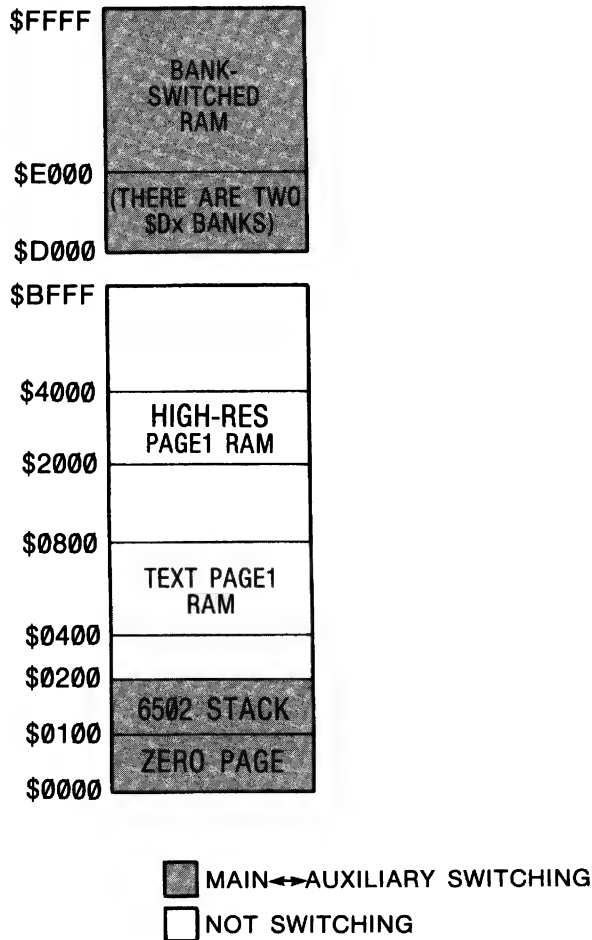


Figure 8-3. The effect of switching ALTZP.

As was mentioned earlier, great care must be taken when using the ALTZP switches to ensure that vital zero page and stack information is not “lost.” All 65C02 operations that affect the stack (this includes PHA, PLA, PHX, PLX, PHY, PLY, PHP, PLP, JSR, and RTS instructions) use the stack that is currently selected by ALTZP, which is not necessarily the stack in main memory.

So, if ALTZP is on and information is stored on the stack in auxiliary memory, don't expect it to be on the stack in main memory when ALTZP is turned off.

Keep in mind that it is extremely important that the value of the 65C02 stack pointer be saved before changing ALTZP and then restored when ALTZP is changed to its original state. If this is not done and the stack pointer is changed while in the other state, then the program will become hopelessly confused and will crash. The following program segment will do the trick:

```

TSX                      ;Put stack pointer in X
STX SAVESP               ; and save it somewhere in memory
STA ALTZPON              ;Turn on ALTZP
.
[execute instructions]
.
STA ALTZPOFF             ;Turn off ALTZP
LDX SAVESP               ;Get original stack pointer in X
TXS                      ; and restore it

```

Any zero page locations that need to be used after ALTZP has been changed will have to be duplicated in the other portion of memory before they can be properly used. To do this, it is necessary to move the contents of zero page into a part of memory that the ALTZP switches do not affect, say \$200 . . . \$2FF, set the appropriate ALTZP switch, and then move this area of memory back down into the new zero page. This process should be repeated when setting ALTZP back to its original position.

The RAMRD and RAMWRT Switches

The RAMRD switches are used to control whether read operations are to use the memory locations from \$200 . . . \$BFFF in main memory or the same locations in auxiliary memory. The RAMWRT switches control write operations for the same area of memory.

If RAMRDON (\$C003) or RAMWRTON (\$C005) is selected, and the 80STOREOFF (\$C000) switch is active, then the entire block of auxiliary memory from \$200 . . . \$BFFF will be selected for reading or writing, respectively. If RAMRDOFF (\$C002) or RAMWRTOFF (\$C004) is selected, then main memory will be selected for reading or writing, respectively, instead. The memory areas that are switched by RAMRD or RAMWRT in each of three different situations are summarized in Figure 8-4.

The area of memory that is affected when the RAMRD and RAMWRT switches are used is slightly different if the switching occurs when 80STOREON (\$C001) is active. As you will recall from Chapter 7, the 80STORE switches are used to define the effect of the //c's PAGE2 switches. If 80STORE is ON, then PAGE2ON (\$C055) and PAGE2OFF (\$C054) are used to select whether the text screen video RAM page (\$400 . . . \$7FF) in auxiliary or main memory is to be selected. In addition, if HIRESON (\$C057) is active, then the PAGE2

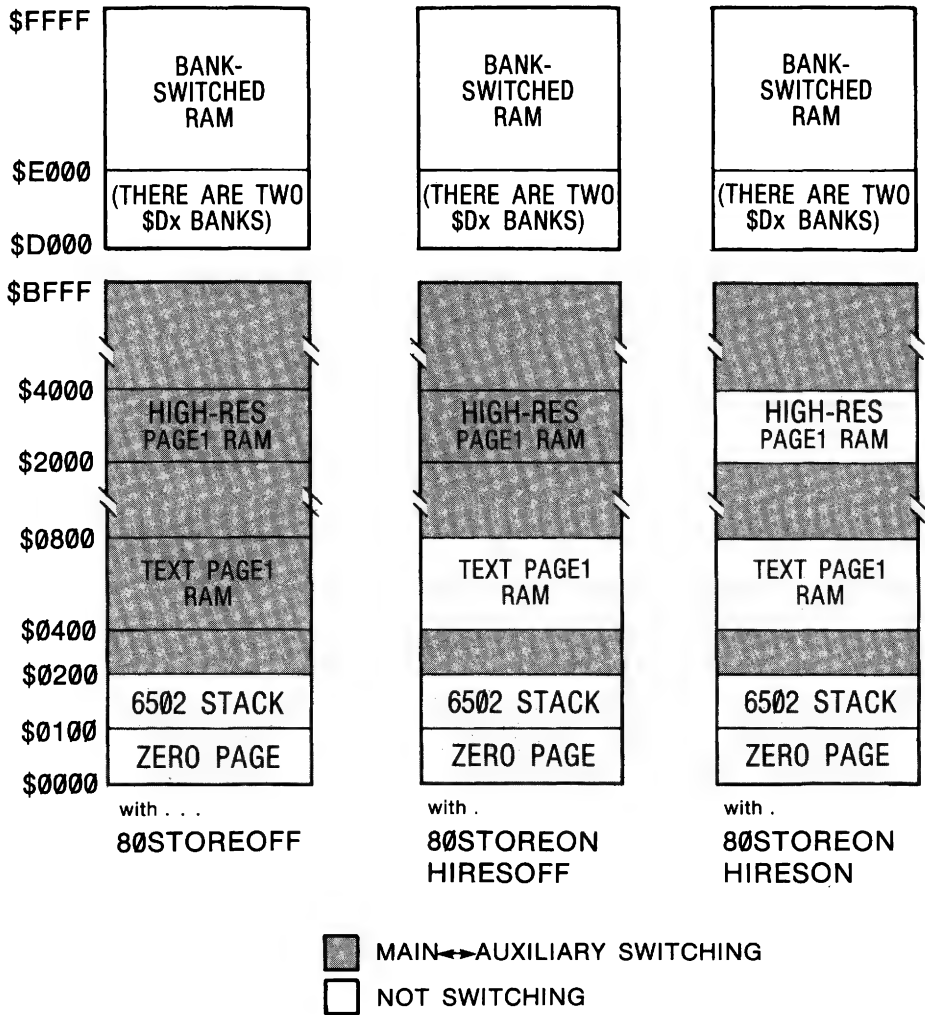


Figure 8-4. The effect of switching RAMWRT or RAMRD.

switches will also select whether the high-resolution graphics screen video RAM page (\$2000 . . . \$3FFF) in auxiliary or main memory is to be selected. The important point to note is that whenever 80STORE is ON, the PAGE2 switches take priority over the RAMRD and RAMWRT switches and so these latter two switches cannot be used to control which of the video RAM areas are active. The effect of switching PAGE2 with 80STOREON is summarized in Figure 8-5.

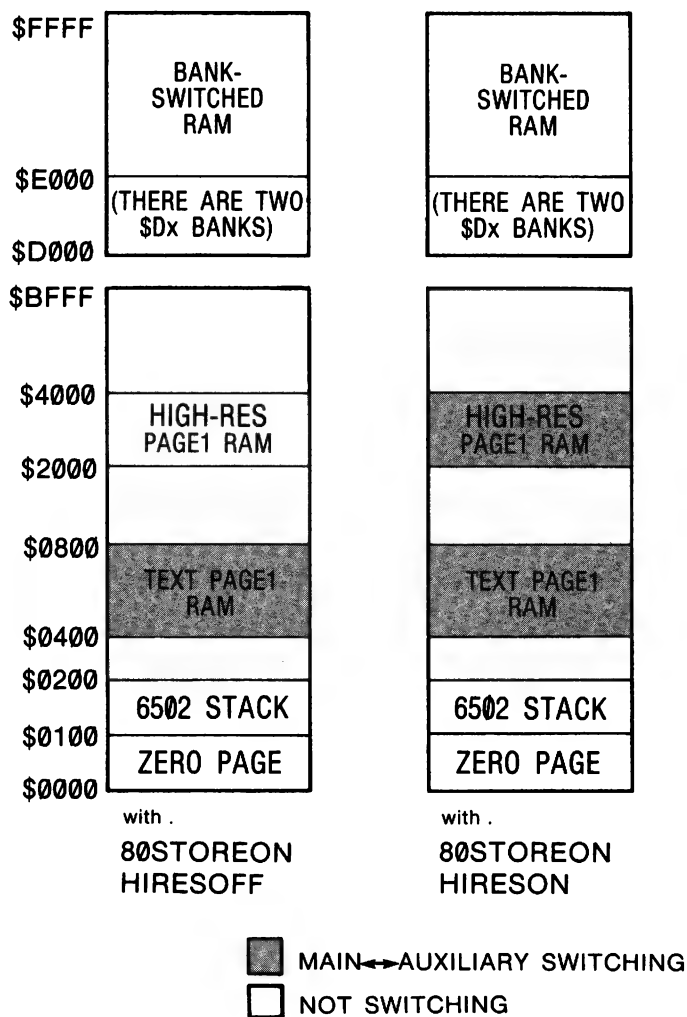


Figure 8-5. The effect of switching PAGE2.

Auxiliary Memory Support Subroutines

The //c has two useful subroutines contained in its system monitor ROM area that facilitate the use of auxiliary memory. These subroutines are called AUXMOVE (\$C311) and XFER (\$C314).

AUXMOVE (\$C311)—Transferring data to and from auxiliary memory

AUXMOVE is used to transfer blocks of data contained within the memory range \$200 . . . \$BFFF from main memory to auxiliary memory or vice versa.

Before using this subroutine, six locations in zero page must be set so that they hold the parameters of the block move. These are summarized in Figure 8-6.

The beginning address of the block to be moved must be stored at locations A1L (\$3C) and A1H (\$3D) and the ending address at A2L (\$3E) and A2H (\$3F). Finally, the destination address must be stored at A4L (\$42) and A4H (\$43). As is usually the case on the //c, the low-order part of each address is stored in the first byte of each zero-page pair.

The state of the 65C02 carry flag is used to tell AUXMOVE the direction of the block move. If the carry flag is set, then the move will be performed from main memory to auxiliary memory. If it is clear, the move will take place in the opposite direction. The state of the carry flag can be set by using the 65C02's CLC (clear carry) and SEC (set carry) instructions. There is no simple way, however, of setting these flags using Applesoft commands; the best that can be done is to call a short machine-language subroutine that clears or sets the carry flag before calling AUXMOVE.

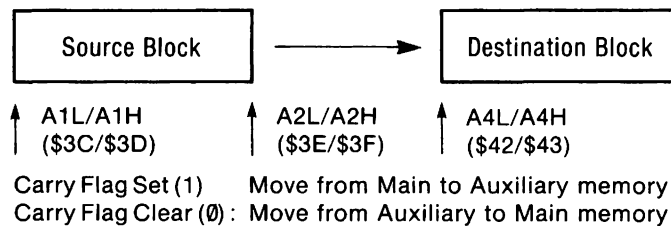


Figure 8-6. AUXMOVE (\$C311) subroutine parameters.

The Applesoft program in Table 8-5 shows how you might transfer an area of memory between main and auxiliary memory. It saves a main-memory high-resolution graphics screen to auxiliary memory and then brings it back again.

The program first installs a short four-byte machine-language program beginning at location 768 (\$300) by POKEing into memory those DATA statement values that appear in lines 120 and 130. These values define the following simple program:

```

SEC
JMP AUXMOVE
  
```

The program then turns on high-resolution graphics and draws a diagonal line on it before setting up the parameters for the block move. In this case, the area of memory to be moved is \$2000 . . . \$3FFF and it will be moved to the area beginning at \$4000 in auxiliary memory. (You shouldn't try to move it to \$2000 . . . \$3FFF in auxiliary memory because if the 80STORE switch is ON—and it will be if the 80-column firmware is being used—and high-resolution graphics are being displayed, then the RAMRD and RAMWRT switches

Table 8-5. AUXMOVE—a program to demonstrate how to move data between main and auxiliary memory.

```
0  REM "AUXMOVE"
100 PRINT CHR$(4);"PR#3"
110 FOR I = 768 TO 771: READ X:
    POKE I,X: NEXT
120 DATA 56: REM "SEC"
130 DATA 76,17,195: REM "JMP $C
    311"
140 HGR : HCOLOR= 3: HPLLOT 10,1
    0 TO 150,150
150 HOME : VTAB 22: PRINT TAB(
    17);"MAIN <---> AUXILIARY ME
    MORY TRANSFER DEMO"
160 HTAB 2: VTAB 23
170 PRINT "PRESS ANY KEY TO SAV
    E THE SCREEN IN AUXMEM: ";: GET
    A$
180 REM SET UP THE PARAMETERS O
    F THE MOVE:
190 POKE 60,0: POKE 61,32: REM
    FROM $2000
200 POKE 62,255: POKE 63,63: REM
    THROUGH $3FFF
210 POKE 66,0: POKE 67,64: REM
    TO $4000 (AUX)
220 CALL 768: REM PERFORM THE M
    OVE
230 HTAB 2: VTAB 23: CALL - 95
    8
240 PRINT "PRESS ANY KEY TO CLE
    AR THE SCREEN: ";: GET A$: HGR
250 HTAB 2: VTAB 23: CALL - 95
    8
260 PRINT "PRESS ANY KEY TO RES
    TORE THE SCREEN FROM AUXMEM:
    ";: GET A$
270 POKE 768,24: REM PUT IN A "
    CLC"
280 REM SET UP THE PARAMETERS O
    F THE MOVE:
290 POKE 60,0: POKE 61,64: REM
    FROM $4000
300 POKE 62,255: POKE 63,95: REM
    THROUGH $5FFF
310 POKE 66,0: POKE 67,32: REM
    TO $2000 (MAIN)
320 CALL 768: REM PERFORM THE M
    OVE
```

that AUXMOVE uses when performing the transfer will not affect this area of memory and no transfer will take place.)

After the screen has been saved to auxiliary memory, you can press a key to clear the screen, and then press another key to restore the line that was drawn on the screen. The line is restored by simply moving the 8K of screen memory that was saved in auxiliary memory back into main memory and not by redrawing the line.

To transfer a block of memory in the opposite direction, the first instruction in the four-byte machine-language subroutine must be changed from SEC to CLC. This is done in line 270 by POKEing 24 into location 768. The number 24 is the value of the CLC instruction.

XFER (\$C314)—Transferring control to a program from main or auxiliary memory

XFER is used to transfer control to a program in either main or auxiliary memory and, at the same time, to select which stack and zero page is to be used when the new program takes over. This is done by setting up certain parameters and executing a JMP (jump) instruction to XFER at \$C314.

As with AUXMOVE, certain parameters and 65C02 flags must be set up before XFER is called. These are summarized in Table 8-6. First of all, the address of the program that is going to take control must be placed at locations \$3ED and \$3EE (low-order byte first). Then, the carry flag must be adjusted to indicate the direction of transfer: it must be set (1) if control is being transferred from a program in main memory to a program in auxiliary memory and clear (0) if transferring control in the reverse direction. Finally, the 65C02 overflow flag must be adjusted to indicate which of the two zero pages and stacks the new program is to use: if it is set (1), then the auxiliary zero page and stack will be used and if it is clear (0), then the main zero page and stack will be used.

Table 8-6. XFER (\$C314) subroutine parameters.

<i>Parameter</i>	<i>Description</i>
Transfer address	\$3ED/\$3EE (low-order byte first). This contains the starting address of the program to which control is to be transferred.
Carry flag	Carry set (1) means “transfer from main to auxiliary memory.” Carry clear (0) means “transfer from auxiliary to main memory.”
Overflow flag	Overflow set (1) means “use auxiliary stack and zero page.” Overflow clear (0) means “use main stack and zero page.”

The CLV (clear overflow) instruction can be used to clear the 65C02 overflow flag to zero. There is no similar command, however, that can be used to set the overflow flag to one. One method of forcing the overflow flag to one is to use the BIT instruction to test any memory location that holds a byte that has a "1" in bit 6. A convenient location to use is \$FF58 because there is an RTS instruction located there and it has an opcode value of \$60.

Of course, before you transfer control to a program in the other memory area, you had better make sure that the program has been loaded there. This is easily done for programs residing in main memory but is a bit more tricky for those residing in auxiliary memory. The easiest way to load a program into auxiliary memory is to use the AUXMOVE subroutine.

Note that the same concerns that were raised about the stack and the stack pointer when discussing the ALTZP switches apply to the use of XFER. It is good practice to save the stack pointer immediately before jumping to XFER and then to restore it if and when a reverse transfer is made. In addition, if the two programs are both using the same stack, care must be taken to avoid overwriting any information that the other program has left on the stack. This is most easily done by saving the whole stack when control is transferred and then restoring it just before returning to the calling program. Alternatively, the two programs should each use a different stack; however, this cannot be done without using two zero pages as well and this may be inconvenient.

Running Co-Resident Programs

As we have seen, the 64K of auxiliary RAM memory is virtually a mirror image of the 64K of main RAM memory. Both of these memory spaces span exactly the same logical addresses, each has its own 65C02 stack and zero page, and each has a 16K area of bank-switched RAM. One important area of difference, however, lies in the use of locations \$400 . . . \$7FF. When in 40-column text mode, only these locations in main memory are used to define the video display; the same locations in auxiliary memory have no effect on the video display. When the 80-column display is active, locations \$400 . . . \$7FF in main memory define the odd-numbered columns in the display while the same locations in auxiliary memory define the even-numbered columns.

The similarities between these two 64K spaces are great enough, however, that it is conceivable that different programs could be loaded into each space and then run independently of one another (well, almost independently of one another). After all, since each program can have its own stack and zero page, there is not a strong temptation for either program to interfere with the other's use of these important areas of memory. The video display will have to be shared, however, for the reasons just given.

The CONCURRENT program in Table 8-7 is a short assembly-language program that allows you to run one of two Applesoft programs that can be

Table 8-7. CONCURRENT—a program to allow switching between Applesoft programs in main and auxiliary memory.

```

1 *****
2 * CONCURRENT *
3 *****
4
5 * (BRUN this program from disk)
6
7 SPSAVE EQU $6      ;Stack pointer save area
8 OLDCSW EQU $7      ;Initial value of CSW (aux. only)
9
10 CSW EQU $36
11
12 * Parameter locations for AUXMOVE:
13 A1 EQU $3C
14 A2 EQU $3E
15 A4 EQU $42
16
17 * Memory switches:
18 STOR800N EQU $C001 ;Don't switch $400...$7FF
19 RAMRDOFF EQU $C002 ;Read main from $200...$BFFF
20 RAMRDON EQU $C003 ;Read auxiliary from $200...$BFFF
21 RAMWRTOF EQU $C004 ;Write main from $200...$BFFF
22 RAMWRTON EQU $C005 ;Write auxiliary from $200...$BFFF
23
24 ALTZPOFF EQU $C008 ;Select main zero page+stack
25 ALTZPON EQU $C009 ;Select auxiliary zero page+stack
26 ALTZP EQU $C016 ;ALTZP status: on if >=$80
27
28 AUXMOVE EQU $C311 ;AUX <--> MAIN move subroutine
29
30 APPLSOFT EQU $E000-1 ;Cold start to Applesoft (less 1)
31

```

```

32 * Monitor initialization subroutines:
33   INIT      EQU   $FB2F
34   HOME      EQU   $FC58
35   SETNORM   EQU   $FE84
36   SETVID    EQU   $FE93
37   SETKBD    EQU   $FE89
38
39   ORG      $2B3
40
41 * Copy SWITCH to auxiliary memory:
42   LDA #<SWITCH
43   STA A1
44   STA A4
45   LDA #>SWITCH
46   STA A1+1
47   STA A4+1
48   LDA #<SWLAST
49   STA A2
50   LDA #>SWLAST
51   STA A2+1
52   SEC
53   JSR AUXMOVE
54
55   STA ALTZPON ;Select aux. zero page + stack
56
57 * Initialize the monitor's auxiliary zero-page usage:
58   CLD
59   JSR SETNORM ;Set normal video
60   JSR INIT    ;Set full-screen text mode
61   JSR SETVID  ;Set for standard 40-column output
62   JSR SETKBD  ;Set for standard 40-column input
63   JSR HOME    ;Clear the screen
64
65 * Redefine output link to keep 80STOREON:
66   LDA CSW

```

(continued)

Table 8-7. CONCURRENT—a program to allow switching between Applesoft programs in main and auxiliary memory (continued).

```

02E0: 85 07 67 STA OLD CSW
02E2: A5 37 68 LDA CSW+1
02E4: 85 08 69 STA OLD CSW+1
70
02E6: A9 44 71 LDA #<NEWOUT
02E8: 85 36 72 STA CSW
02EA: A9 03 73 LDA #>NEWOUT
02EC: 85 37 74 STA CSW+1
75
76 * Initialize auxiliary memory stack:
02EE: A9 DF 77 LDA #>APPLSOFT ;Set up a return to the cold
02F0: 8D FF 78 STA $1FF ; start entry point for
02F3: A9 FF 79 LDA #<APPLSOFT ; Applesoft the first time
02F5: 8D FE 80 STA $1FE ; you enter auxiliary memory
02F8: A2 FD 81 LDX #$FD ;Set up initial stack pointer
02FA: 86 06 82 STX SPSAVE ; and save it in aux. memory
83
02FC: 8D 08 C0 84 STA ALTZPOFF ;Select main zero page + stack
85
02FF: 60 86 RTS
87
88 * SWITCH is used to move between aux. and main:
0300: 8E 41 03 89 STX XSAVE ;Save X, Y, A, P, S
0303: 8C 42 03 90 STY YSAVE
0306: 8D 40 03 91 STA ASAVE
0309: 08 92 PHP
030A: 68 93 PLA
030B: 8D 43 03 94 STA PSAVE
030E: BA 95 TSX
030F: 86 06 96 STX SPSAVE
0311: 8D 01 C0 97 STA STOR800N ;Don't switch video RAM

```

```

0314: AD 16 C0 98      LDA      ALTZP
0317: 30 1B 99      BMI      TOMAIN
0319: 8D 09 C0 100    STA      ALTZPON
031C: 8D 03 C0 101    STA      RAMRDDN
031F: 8D 05 C0 102    STA      RAMWRTON
0322: A6 06 103    RESTORE  LDX      SPSAVE
0324: 9A 104      TXS
0325: AE 41 105    LDX      XSAVE
0328: AC 42 106    LDY      YSAVE
032B: AD 43 107    LDA      PSAVE
032E: 48 108      PHA
032F: AD 40 109    LDA      ASAVE
0332: 28 110      PLP
0333: 60 111      RTS
0334: 8D 08 C0 112    STA      ALTZPOFF
0337: 8D 02 C0 113    STA      RAMRD0FF
033A: 8D 04 C0 114    STA      RAMWRT0F
033D: 4C 22 03 115    JMP      RESTORE
                                TOMAIN
116      ASAVE      DS      1
117      XSAVE      DS      1
118      YSAVE      DS      1
119      PSAVE      DS      1
120
121
122      * The following new input subroutine is really needed only
123      * when Applesoft is first initialized. When Applesoft is
124      * initialized, it writes to $C000, thus turning 80STOREOFF
125      * and preventing the program in auxiliary memory from
126      * using the 40-column video RAM (in main memory).
0344: 8D 01 C0 127    NEWOUT  STA      STOR800N
0347: 6C 07 00 128    JMP      (OLDCSW)
129
130      SWLAST      EQU      *

```

loaded into memory at the same time, one in main memory and the other in auxiliary memory, and to easily switch between the two programs. It must be activated by using the BRUN command to load and execute it directly from diskette; you must be in standard 40-column mode before doing this.

The first thing that CONCURRENT does is to copy its SWITCH and NEWOUT subroutines from main to auxiliary memory by using the AUXMOVE subroutine. The SWITCH subroutine is responsible for transferring control from main to auxiliary memory and vice versa, and so a copy of it must be stored in both these memory areas to ensure that it is always available. There is one other important reason for duplicating SWITCH in this way. Part way through the subroutine, the area of memory (main or auxiliary) that is currently active will be turned off and replaced by the other, thus causing the current copy of SWITCH to temporarily vanish. This would normally cause the system to hang because the instruction at the next address at which SWITCH resumes executing after switching would no longer be available and the program would behave unpredictably. If SWITCH is present at exactly the same locations in the other area of memory, however, then one copy will always be active and no problems will be encountered.

After CONCURRENT has moved SWITCH to auxiliary memory, it enables the zero page and stack in auxiliary memory (ALTZPON) and then calls five system monitor initialization routines (SETNORM, INIT, SETVID, SETKBD, and HOME) that will cause the auxiliary zero page to be properly initialized so that system monitor I/O subroutines will work properly.

The next task that CONCURRENT performs is to redefine the standard character output subroutine by storing the address of the NEWOUT subroutine at the CSWL/CSWH (\$36/\$37) output link in auxiliary memory. The NEWOUT subroutine must be used to handle output because of a complication that arises when Applesoft is first initialized in auxiliary memory.

When Applesoft is first initialized, it determines how much RAM memory is installed in the //c by storing and reading numbers at the first location of each memory page (beginning with page \$08) until it finds that the number read is not the same as the number stored. When such a discrepancy occurs, then a non-RAM location must have been reached.

On the //c, the first non-RAM address written to will be \$C000, which is the first address in the //c's I/O memory space. Unfortunately, this has the side effect of turning off the 80STORE soft switch that resides at that location. This means that if the RAMRD and RAMWRT switches are on (that is, auxiliary memory from \$200 . . . \$3FF and \$800 . . . \$BFFF is active), then the auxiliary memory space from \$400 . . . \$7FF will be active as well. (Remember that this same space in main memory—which represents the video RAM for the 40-column text screen—will remain active if 80STORE is on.) This auxiliary memory space has no effect on the 40-column screen display, however, and so the screen display will not change when attempts are made to update

it by calling the standard video subroutines (that only affect the currently active \$400 . . . \$7FF space).

If we could turn the 80STORE switch on before Applesoft tries to perform its first video operation after initialization (the displaying of its “]” prompt symbol), then we could avoid the problem of having the “wrong” \$400 . . . \$7FF space active. This is done by replacing the standard output subroutine with the nearly identical NEWOUT subroutine. In fact, the only difference is that NEWOUT first writes to 80STOREON (\$C001) to ensure that the video RAM area from \$400 . . . \$7FF in main memory will be active.

Initialization of the Auxiliary Stack

After the new output link address is set up in auxiliary memory, CONCURRENT initializes the auxiliary stack by placing the address, less 1, of the cold start entry point to Applesoft (\$E000) at the first two stack locations, \$1FF and \$1FE. The high-order part of this address is stored at \$1FF and the low-order part at \$1FE. After this has been done, the value \$FD is stored at SPSAVE, a temporary storage location. The first time that auxiliary memory is enabled by calling SWITCH, the 65C02 stack pointer register will be loaded from SPSAVE, meaning that when the RTS is executed at the end of the SWITCH subroutine, control will be returned to \$E000, the Applesoft cold start entry point. The subroutine at \$E000 takes care of initializing Applesoft in auxiliary memory by setting up all its program and data pointers that are contained in zero page.

The last thing that CONCURRENT does is re-enable zero page and the stack in main memory and then end. At this point the //c is configured in such a way as to allow you to easily switch between programs in main and auxiliary memory.

Using CONCURRENT

CONCURRENT is simple to use. Whenever you want to leave main memory and resume running the program in auxiliary memory, or vice versa, you must activate the SWITCH subroutine by entering a CALL 768 command. This subroutine determines which bank of memory is active (by examining the status of the ALTZP switch) and then enables the other bank of memory from \$0000 . . . \$BFFF (except for the \$400 . . . \$7FF video RAM area) for reading and writing by adjusting the ALTZP, RAMRD, and RAMWRT switches accordingly. The \$400 . . . \$7FF video RAM area in main memory is kept active by writing to 80STOREON (\$C001) before accessing the RAMRD and RAMWRT switches.

When the SWITCH subroutine ends, it executes an RTS instruction that instructs the 65C02 to return to the address (plus 1) that is stored on the top of the stack. Unless some tricky programming is done, this address is that of

the instruction immediately following the JSR instruction that called the subroutine. Such is not the case, however, when calling SWITCH because just before its RTS instruction is executed, the other stack is reactivated and its stack pointer is set equal to the value it had when SWITCH was last called. What this means is that as soon as SWITCH is called, the //c begins executing those instructions right after the CALL 768 that activated the switch in the first place.

To see a simple example of how CONCURRENT works, first install SWITCH by executing CONCURRENT. Then enter the following Applesoft program:

```
100 IF PEEK (49152) = 155 THEN
      POKE 49168,0: CALL 768
200 PRINT "MAIN MEMORY": GOTO 100
```

and RUN it. This program doesn't do much but continuously print out "MAIN MEMORY" on the screen. However, the program is constantly monitoring the keyboard for an ESC character in line 100. If ESC is pressed, then the keyboard strobe is cleared (POKE 49168,0) and then SWITCH is called by executing a CALL 768 command.

When SWITCH is called for the first time, you will be put into Applesoft direct mode in auxiliary memory. While you are there for the first time, enter this program:

```
100 IF PEEK (49152) = 155 THEN
      POKE 49168,0: CALL 768
200 PRINT "AUXILIARY MEMORY": GOTO 100
```

This is the same as the previous program, except that it prints out "AUXILIARY MEMORY." Now type RUN to start this program and then press the ESC key. As soon as ESC is pressed, you will switch back to main memory and the program there will resume executing right where it left off and will start printing "MAIN MEMORY." By pressing ESC again and again, you can see that you are indeed switching between the two programs!

Limitations of CONCURRENT

The major limitation of CONCURRENT is that the program running in auxiliary memory cannot use any ProDOS commands. Although a copy of ProDOS could be transferred to auxiliary memory and used by the program running there, several problems could arise that would be difficult to solve in software. For example, special "lockout" flags would have to be used to prevent one program from modifying a file until the other had finished using it. If this was not done, then the data in the file could easily become scrambled. Rather than complicate the CONCURRENT program and obscure its usefulness as an example of how to use main and auxiliary memory, no attempt has been made to allow the program in auxiliary memory to use ProDOS.

Other problems arise because the two programs must share the same video

screen. This means that information placed on the screen by one program could easily be overwritten by the other. One solution to this problem is to define nonoverlapping text windows for each program by modifying the window parameters held in zero page. (See Chapter 7.)

Since auxiliary memory is initialized by installing the standard 40-column input and output subroutines (by calling SETVID and SETKBD), you should not enter auxiliary memory when 80-column mode is active. If you wish to use CONCURRENT with an 80-column display, the "JSR SETVID" and "JSR SETKBD" instructions should be replaced by a "JSR \$C300" instruction; the latter instruction takes care of installing the I/O subroutines that support the 80-column display. Alternatively, you can simply turn 80-column mode on by entering PR#3 after installing CONCURRENT.

Finally, you should note that you must not write to 80STOREOFF (\$C000) while in auxiliary memory. If this is done, then the auxiliary memory space from \$400 . . . \$7FF will become active and, as explained above, you will not be able to display anything on the video screen.

Further Reading for Chapter 8

Standard reference works . . .

80-Column Text Card Manual, Apple Computer, Inc., 1982.

Extended 80-Column Text Card Supplement, Apple Computer, Inc., 1982.

Note: the above two reference manuals are written for the optional 80-column text card used on the Apple //e only. However, much of the information in them is directly applicable to the built-in 80-column display on the //c as well.

On uses for auxiliary memory . . .

D.C. Johnson, "Using Auxiliary Memory in the //e", *Apple Assembly Line*, August 1983, pp. 2–12. Another program to switch between main and auxiliary memory.

9

The Speaker

In this chapter, we will examine another built-in I/O device that the `//c` supports: the speaker.

The `//c`'s speaker can be used to add the dimension of sound to programs. In many cases, this simply means that you will hear a short (but suitably aggravating) beep whenever you make an error. However, some programs, notably educational software and games, exercise the speaker in much more dramatic ways to generate complex sound effects and recognizable musical patterns that tend to dramatically liven up these types of programs. We will look at the techniques used to generate music later in this chapter.

You can make the `//c`'s speaker beep at you by entering or printing the ASCII "bell" character (ASCII code 7). This can be done by pressing [control-G] on the keyboard or by printing `CHR$(7)` from Applesoft. (If you don't hear anything, turn up the volume by adjusting the control wheel on the left side of the `//c`). With appropriate software, the speaker can also be used to generate music and sound effects and even to reproduce (though crudely) the human voice.

The sounds that the speaker generates are caused by the in and out movement of the speaker cone; the frequency (also called the pitch) of a sound is the same as the frequency of the cone's movement. The position of the cone is controlled by a voice coil and a permanent magnet located near the base of the cone. When this coil is turned on, the cone moves out and when it is turned off, the cone moves in. Thus, you can select the frequency of the tone to be emitted merely by switching this coil on and off at the desired frequency.

There is one special I/O memory location reserved for the speaker that allows you to control it in this way. As indicated in Table 9-1, this is `SPEAKER` (`$C030`). This is yet another soft switch location; each time that it is read (using Applesoft's `PEEK` or an assembler's `LDA`) the state of the speaker changes from off to on (if it was last off) or from on to off (if it was last on).

Generating Musical Notes

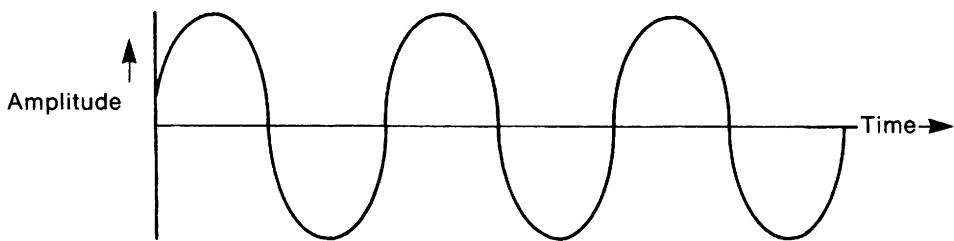
Let's take a close look at how you can use the `//c` to generate musical notes. First, recognize that the sound wave generated by a single musical note is

Table 9-1. Speaker I/O memory location.

<i>Address</i>		<i>Symbolic Name</i>	<i>Description</i>
<i>Hex</i>	<i>(dec)</i>		
\$C030	(49200)	SPEAKER	Speaker output. Reading this location toggles the state of the speaker.

merely a smoothly varying sine wave, as shown in Figure 9-1 (a). Since, however, we can only turn the //c's speaker on or off (that is, we cannot smoothly vary the amplitude of its output), we can only generate square waves like the one shown in Figure 9-1 (b). For most frequencies, however, this square wave is an acceptable approximation of its sine wave equivalent and the sound that is generated is close to what you would normally expect to hear.

(a) Sine wave for pure tone.



(b) Square wave approximation of pure tone.

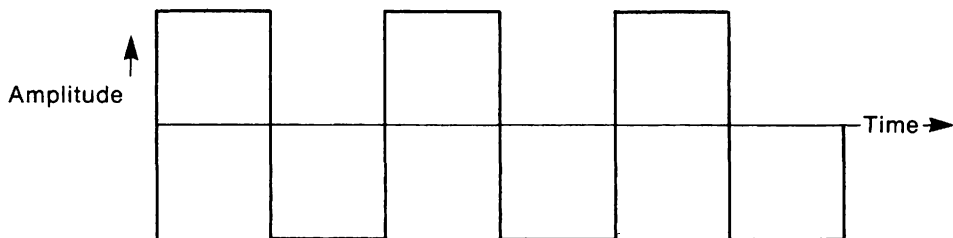


Figure 9-1. Sine waves and square waves.

Before a specific note can be generated, you will have to know its frequency (or "pitch"). Table 9-2 contains a list of two octaves of musical notes from Low "C" through Middle "C" to High "C", their frequencies on the standard

Even-Tempered Scale in hertz (cycles/second), and their periods. The period is equal to the time it takes to finish one complete sinusoidal cycle and is equal to the reciprocal of the frequency.

To generate the waveform for any note, the speaker must be turned on for one-half of its period and off for the other half. Given this information, the procedure to follow for generating a note is as follows:

- Turn the speaker on
- Wait one-half period
- Turn the speaker off
- Wait one-half period
- Return to step 1

Table 9-2. Frequencies and periods of musical notes on the even-tempered scale.

<i>Note</i>	<i>Frequency (Hz)</i>	<i>Period (μsec)</i>	<i>HALFTIME*</i>
C (Low "C")	131	7,634	112
C#	139	7,194	106
D	147	6,803	100
D#	156	6,410	94
E	165	6,061	89
F	175	5,714	84
F#	185	5,405	79
G	196	5,102	75
G#	208	4,808	71
A	220	4,545	67
A#	233	4,292	63
B	247	4,049	60
C (Middle "C")	262	3,817	56
C#	277	3,610	53
D	294	3,401	50
D#	311	3,215	47
E	330	3,030	45
F	349	2,865	42
F#	370	2,703	40
G	392	2,551	38
G#	415	2,410	35
A (Concert "A")	440	2,273	33
A#	466	2,146	32
B	494	2,024	30
C (High "C")	523	1,912	28

*See text.

Since the status of the speaker toggles between on and off every time you access its soft switch at **SPEAKER** (\$C030), you can simplify this flowchart by removing steps 3 and 4.

The above procedure must be repeated for the duration of the note; if you are playing a note from a piece of music, this duration will depend on the type of note that is being played (a whole note, half-note, quarter-note, and so on) and the tempo of the music.

Table 9-3 shows the **NOTE** program that uses the //c's speaker to produce a note of a specified frequency and duration. This program toggles the speaker whenever the **X** register, which at the beginning of every tone cycle contains a code number related to the period of the note, is reduced to zero by successive **DEX** instructions. The **X** register is reduced by one unit every $34 \times \text{HALFTIME}$ microseconds, where **HALFTIME** is this code number and 34 happens to be the length of an internal software delay loop that has been used. The code number is simply equal to the number of 34-microsecond loops that must be performed before one-half of the period of the note elapses. It can be calculated by dividing one-half of the period time (in microseconds) by 34. For example, the value of **HALFTIME** for an "A" note (440 Hz) would be equal to 1136 (one-half its period in microseconds) divided by 34, which is equal to 33. In this way, you can easily calculate the **HALFTIME** values for all the other notes that you may wish to generate; they are listed in Table 9-2 for your convenience.

NOTE also allows you to specify the duration of the note to be played by adjusting the **LENGTH** constant. A temporary value of **LENGTH**, called **LTEMP**, is decremented each time the program executes 255 of the aforementioned 34-microsecond loops, that is, once every 8670 microseconds. Thus, to play a note for one second (1,000,000 microseconds), **LENGTH** would be set equal to $1,000,000/8670$, or 115.

The loop time in the **NOTE** program has been calculated by determining exactly how many 65C02 machine cycles take place between successive reductions of the loop counters (that control the frequency and duration of the note) and multiplying that number by the period of the 65C02 microprocessor's clock. Since the //c's clock is operating at about 1 MHz, it turns out that the loop time (in microseconds) is simply equal to the number of machine cycles needed to perform the instructions in the loop. To calculate the total number of machine cycles being performed in the loop, you must first determine what instructions are being executed in the loop and then add up their individual cycle times. The cycle times for each 65C02 instruction are listed in Appendix II. Note that the number of cycles depends not only on the particular instruction being executed but also on the addressing mode that is being used by that instruction.

It should be obvious by now that because of the meticulous timing loops that music programs require to produce precise frequencies, it is really not possible to create quality music by directly accessing **SPEAKER** (\$C030) using

the Applesoft PEEK statement and FOR/NEXT loops. Applesoft delay times simply cannot be adjusted as finely as can assembler delay times and, even if they could be, they could actually change depending on the location of the loop in the program. So stick to assembly language if you want to create music. Applesoft programs can be used, however, to POKE frequency and duration information into an assembly-language program's data area and to CALL the assembly-language program. We will see how to do this next.

Generating Music

Now that we have written a program to generate one musical note, it will be almost trivial to develop a program that actually plays a short tune. All we have to do is link single notes together in the orders, and for the durations, dictated by the sheet music for the tune.

Consider the Applesoft SONG program in Table 9-4. This program contains several DATA statements that contain the HALFTIME and LENGTH values needed by NOTE for each of the notes in the first part of the theme from the television series "M*A*S*H." The LENGTH values have been calculated by assuming that a whole note has a duration of one second; if this is the case, then LENGTH=115, as explained earlier. To play the tune defined by the DATA statements, first ensure that the NOTE program has been saved to diskette and then enter the Applesoft RUN command. SONG plays the tune by executing an Applesoft FOR/NEXT loop that reads the HALFTIME and LENGTH values for a note, POKES them into the NOTE program data area, and then CALLs the NOTE program to generate the tone. After all the notes have been played in this way, the program ends.

You can easily play your own favorite song by translating its notes into HALFTIME and LENGTH values and placing these values into the DATA statements of the SONG program. The last pair of values in the DATA statements must be zeros so that SONG will know when all the notes have been read.

You may well be wondering whether you can play chords of music, that is, more than one note at once, in order to improve the quality of the sound that is generated. The short answer is "yes, you can!" but the software required to do this is much more complex. For example, to play two notes at once, you would have to intertwine two timing loops, one for each note, and you would have to ensure that the speaker was being toggled at the proper rate for each note. This is not an impossible feat to be sure, but it is left as an exercise for the more interested reader.



Table 9-3. NOTE—a program to play a musical note on the //c's speaker.

```

1  *****
2  * NOTE *
3  *****
4
5  SPEAKER EQU $C030      ;Speaker I/O location
6
7  ORG $300
8
9  HALFTIME DFB 33         ; = (1/frequency)/(2*34)
10 LENGTH DFB 29          ;Duration in units of 34*255 usec
11
12 NOTE LDY #255           ;The program starts here
13 LDA LENGTH
14 STA LTEMP
15 NOTE1 LDX HALFTIME      ;X contains the length of the note
16 LDA SPEAKER            ;Toggle the speaker
17 JMP STALL1
18 STALL NOP
19 STALL NOP
20
0300: 21
0301: 1D
0302: A0 FF
0304: AD 01 03
0307: 8D 2F 03
030A: AE 00 03
030D: AD 30 C0
0310: 4C 1A 03
0313: EA
0314: EA

```

These NOPs compensate for
; branches to NOTE1 from line 37

Table 9-4. SONG—a program that plays a song on the //c's speaker.

```
0  REM "SONG"
100 PRINT CHR$(4);"BLOAD NOTE
   "
110 READ HT: READ LN: REM READ
    HALFTIME AND LENGTH
120 IF HT = 0 AND LN = 0 THEN 1
    60
130 POKE 768,HT: POKE 769,LN
140 CALL 770: REM PLAY THE TONE

150 GOTO 110: REM AND GET NEXT
    NOTE
160 END
1000 REM NAME THAT TUNE!!
1010 DATA 63,29,67,29,63,29,67,
    29,63,29,67,29,75,58
1020 DATA 67,29,75,29,67,29,75,
    29,67,29,75,29,84,29,67,29,7
    5,29,84,29,75,29,84,29
1030 DATA 75,29,84,29,89,29,75,
    29,84,29,89,29,84,29,89,29,8
    4,29,75,29,67,58
1040 DATA 67,58,56,29,50,29,56,
    29,50,29,56,29,50,29,56,58,5
    6,29
1050 DATA 50,29,56,29,50,29,56,
    29,50,29,56,58,56,29,67,29,5
    6,29,50,29,42,29
1060 DATA 38,29,42,29,50,29,56,
    29
1070 DATA 50,115,50,58,50,29,56
    ,29,67,29,56,29,50,29,42,29
1080 DATA 38,29,42,29,50,29,56,
    29,50,115,50,58
1090 DATA 0,0: REM END OF DATA
    MARKER
```

Further Reading for Chapter 9

On the speaker . . .

“Apple Noises and Other Sounds”, *Apple Assembly Line*, February 1981, pp. 2–9. Generating sound effects using the speaker.

B.C. Detterich, “Apple Free Speech”, *Call -A.P.P.L.E.*, September 1981, pp. 9–14. Using the Apple II speaker to generate voice and sound.

J.H. Bender, “Pitch and Rhythm on the Apple”, *Call -A.P.P.L.E.*, June 1982, p. 15. More on music for the Apple.

B. Sander-Cederlof, “Your Apple Can Talk”, *Apple Assembly Line*, November 1982, pp. 2–9

10

Mouse and Game Controller Input

In this chapter we will be examining in detail how to use the I/O connector located on the back panel of the //c at the far left (as viewed from the back end). This port is primarily used to interface two types of related input devices to the //c: the Apple mouse and the game controller. These two devices are related in that they both provide positional information to the //c. That is, it is possible for the //c to determine where a mouse is located on the desktop or how far a game controller knob has been rotated.

Unfortunately, you cannot simultaneously connect a game controller and a mouse to the //c because both devices use the same connector. Fortunately, however, this limitation is recognized by the //c's built-in firmware and it is possible to read positional information from the mouse as if it was a real game controller. This means that you can use the mouse with most software that is designed for use with standard game controllers. We'll see how to do this later in the chapter.

The Apple Mouse

What is a mouse, anyway? Well, it's simply another input device that can be used to transmit useful information to the //c. The information that it transmits, however, is not like the ASCII codes that a keyboard delivers, or the 8-bit bytes of data that a disk drive delivers, but rather is information relating to the position of the mouse on a two-dimensional X-Y plane (that's math talk for desktop). Software can easily be written to translate the mouse's position into a coordinate on the video screen so that a cursor or pointer can be displayed that keeps in step with the movement of the mouse.

One of the main advantages of mouse-based software is that you usually don't have to use the keyboard in order to select options or enter commands. Instead, all you have to do is move the mouse until the screen pointer is positioned in the area of the screen that contains the symbolic representation of the command you want performed (either a pictorial icon or a command

phrase), and then click the mouse button. People who can't type like this technique a lot! As all users of the Apple Macintosh computer know, mouse-driven software can also be incredibly easy to learn.

As far as the //c is concerned, the mouse is a port 4 device. This means that the firmware that controls it can be accessed by using the Applesoft PR#4 and IN#4 commands. We'll see examples of this later on in this chapter.

How the Mouse Works

The mouse is quite an interesting device. The actual mouse unit itself is a small, rectangular, box-like structure with a flat push button on top. The status of the mouse button ("pressed" or "not pressed") can be monitored by reading a special I/O location in the //c's memory. Electrical signals from the mouse unit are transmitted to the mouse connector at the back of the //c through a long cord (the "tail").

Inside the mouse is some special electronic circuitry that is responsible for translating the motion of the mouse into a stream of electrical signals that the //c can decode and transform into horizontal (X) and vertical (Y) coordinate information. This information can then be used to position a cursor or a pointer on the video screen.

If you turn the mouse unit over, you will see a small rubber ball poking out of a cavity inside. When the mouse is moved across the table, this ball rotates; it is this rotational movement that allows the mouse to transfer positional information to the //c.

Inside the cavity that the ball occupies are two small, cylindrical rollers that have been arranged at a 90 degree angle to one another so that one will turn when the mouse is moved in an up/down (Y) direction and the other will turn in response to left/right (X) movement. Of course, if the mouse is moved diagonally, then both rollers will be set in motion.

Passing through the centre of each roller is a long metal axle, one end of which is connected to the centre of a small flat disk that has several slits symmetrically arranged around its circumference. As the roller turns, so does the axle and the disk. The movement of the disk will cause a narrow beam of light that is aimed from a tiny infrared light source on one side of the disk to a photoreceptor on the other side to alternately pass through unaffected or be blocked, depending on whether or not a slit is in the path of the beam. Thus, the electrical output from the photoreceptor will either be 1 (light beam not blocked) or 0 (light beam blocked) and a square wave signal will be generated when the mouse is moved. This square wave is passed down the mouse's tail to the //c where it can be used to generate a 65C02 IRQ interrupt on each rising edge (or, alternatively, each falling edge) of the square wave. The two square waves that are generated (one for each roller) are called X0 and Y0.

When an interrupt is generated by movement of the mouse, you can determine the axis of movement (X or Y, or both) by examining two I/O status memory locations. Electronic circuitry inside the mouse also provides information on the direction of movement along each axis (up or down, left or right) and this information can be read by examining two other I/O memory locations.

The *//c* has a special built-in interrupt-handling subroutine that responds to mouse interrupts and is responsible for monitoring the mouse's position and button status. Whenever a program needs updated mouse information, it simply calls a built-in firmware subroutine. We will be looking at how to do this later in this chapter.

Mouse Operating Modes

The mouse can be operated in one of four fundamental modes:

- Passive (or transparent) mode
- Movement interrupt mode
- Button interrupt mode
- Movement or button interrupt mode

A mode is selected by sending a command to the port 4 firmware that is built-in to the *//c*. We will see how to do this in later sections of this chapter. (In summary, it involves calling a subroutine called SETMOUSE with a mode control byte in the accumulator.)

If you will be using one of the interrupt modes, then you must make absolutely sure to install an interrupt-handling subroutine first. As we saw in Chapter 2, this can be done by loading the subroutine into memory and then placing its starting address in locations \$3FE and \$3FF (low-order byte first). If you don't do this, you will likely enter limbo when the first interrupt occurs.

You should note that mouse interrupts are synchronized with the *//c*'s vertical blanking (VBL) interrupt signal that we briefly discussed in Chapter 7. This means that a mouse interrupt signal will not actually be recognized until such time as the video display starts a vertical retrace operation (this happens every 1/60 of a second).

Passive (Transparent) Mode

In passive mode, the mouse position and button status are constantly being updated "behind the scenes" and no interrupts are passed through to a user-installed interrupt-handling subroutine when they change. It is the responsibility of the program that is running to periodically read the mouse coordinates and button status to see if they have changed.

You should note, however, that even in passive mode, mouse interrupts are still occurring since this is the only way in that the //c can properly track mouse motion. These interrupts are serviced internally, however, and are not passed through to the user.

Movement Interrupt Mode

If movement interrupt mode is selected, the //c will pass control to your own interrupt-handling subroutine whenever the mouse is moved. This subroutine can identify the source of the interrupt as the mouse button by calling the READMOUSE subroutine and then checking to see if bit 1 of MOUSTAT (\$77C) is set to 1. The READMOUSE subroutine and the MOUSTAT status location will be discussed in detail later in this chapter.

Button Interrupt Mode

At the beginning of this chapter, when we reviewed how the mouse works, we mentioned that the mouse generates interrupts whenever it is moved. We did not mention, however, anything about a button interrupt signal. There was a very good reason for this, of course: the mouse button has not been designed to generate interrupts! How then can a button interrupt mode be supported? With software mirrors of course!

When button interrupt mode is selected, the //c automatically enables the VBL interrupts and when such an interrupt occurs, the status of the mouse button is examined. If the button is being pressed, the //c passes control to your own interrupt-handling subroutine that can identify the source of the interrupt as the mouse button by calling the READMOUSE subroutine and then checking to see if bit 2 of MOUSTAT (\$77C) is set to 1.

Movement or Button Interrupt Mode

This mode is simply a combination of the movement interrupt mode and the button interrupt mode. In this mode, a 65C02 interrupt will be passed through whenever the mouse button is pressed or the mouse is moved.

Vertical Blanking Interrupts

It is also possible to interrupt the //c every 1/60 second by enabling the vertical blanking (VBL) interrupt signal. As we saw in Chapter 7, this interrupt is generated immediately after the video display has been refreshed when the video electron beam begins retracing to the top left-hand corner of the screen. The //c allows you to enable or disable VBL interrupts when any other mouse mode is selected or even when the mouse is off.

The Mouse and Applesoft

Perhaps the simplest way to become acquainted with the mouse is to learn how it can be controlled from an Applesoft program. This is simple because there are really only three commands with which we need be concerned. These commands perform the following functions:

- Turning the mouse on
- Turning the mouse off
- Reading the mouse coordinates and the mouse button

If you want the mouse to perform fancier tricks, such as interrupting the system whenever it is moved or its button is pressed, then you will have to rely on assembly language instead. Let's leave that particular subject for later, however. In the meantime, let's look at the three mouse-control functions that can be executed directly from Applesoft. These functions are summarized in Table 10-1.

Turning the Mouse On

The mouse can be turned on by executing an Applesoft program line that looks something like this:

```
100 PRINT CHR$(4);"PR#4": PRINT CHR$(1)
```

Pretty simple, isn't it? This Applesoft statement simply redirects output to the port 4 firmware that controls the mouse interface (with a PR#4 command) and then sends ASCII code 1 (CONTROL-A) to the firmware. This code is interpreted by the mouse firmware to mean "turn on the mouse." Only when the mouse is on can its position and button status be read.

After the mouse has been turned on in this way, it is a good idea to execute a line such as this:

```
200 PRINT CHR$(4);"PR#0"
```

in order to ensure that any subsequent output gets sent to the video screen instead of the mouse firmware. (Change that PR#0 to a PR#3 if you are in 80-column mode.)

You can also turn the mouse on by entering commands from the keyboard when in Applesoft direct mode (that is, when the "]" prompt symbol is being displayed). To do this, first enter

```
PR#4 [return]
```

and then press [control-A] followed by the [return] key. After you've done this, enter

Table 10-1. Controlling the mouse from Applesoft.		
<i>Mouse ON:</i>		
100 PRINT CHR\$ (4);"PR#4":PRINT CHR\$(1)		← program line
PR#4 [return] [control-A] [return] PR#0 [return]		← from the keyboard
<i>Mouse OFF:</i>		
100 PRINT CHR\$ (4);"PR#4":PRINT CHR\$(0)		← program line
PR#4 [return] [control-@] [return] PR#0 [return]		← from the keyboard
<i>Mouse READ:</i>		
100 PRINT CHR\$ (4);"IN#4"		
200 INPUT " ";X,Y,B		
X is the horizontal mouse position (0...1023). Y is the vertical mouse position (0...1023). B is the button status code (see Table 10-2).		

```
PR#0 [return]
```

or

```
PR#3 [return]
```

to redirect output to the video screen.

Turning the Mouse Off

The procedure to follow to turn the mouse off is as simple as the one used to turn it on. To turn it off, execute the following program line:

```
100 PRINT CHR$(4);"PR#4": PRINT CHR$(0)
```

The mouse firmware interprets ASCII code 0 (the null) as the mouse off command. The above line should always be followed by one that redirects output to the video display:

```
200 PRINT CHR$(4);"PR#0": REM 40-  
COLUMN DISPLAY ON
```

or

```
200 PRINT CHR$(4);"PR#3":REM 80-  
COLUMN DISPLAY ON
```

The series of keyboard commands that can be entered to turn off the mouse are as follows:

```
PR#4 [return]  
[control-@] [return]  
PR#0 [return] or PR#3 [return]
```

Reading the Mouse

Now we know how to turn the mouse on and off. This information isn't much good to us, however, unless we also know how to read the data that defines the mouse's position and the status of its button. Read on to find out how to do this.

Before valid mouse data can be read, the mouse must be turned on as described above. When the mouse is on, its data can be read by selecting the mouse firmware in port 4 for input by executing a program line like this:

```
100 PRINT CHR$(4);"IN#4"
```

Once this has been done, the current X and Y coordinates of the mouse, and the status of its button, can be read by executing the following statement:

```
200 INPUT " ";X,Y,B
```

where X, Y, and B represent any three Applesoft numeric variables. You should note two important features of this INPUT statement:

- The null string in the INPUT statement has been included to prevent the display of the question mark prompt that an INPUT statement usually uses.
- The three variables are all part of the same INPUT statement. The variables cannot be read in with separate INPUT statements.

After the INPUT statement has been executed, the mouse's horizontal (X) position will be in variable X, the vertical (Y) position in variable Y, and the mouse button status code in variable B.

The X and Y coordinates of the mouse will be in the range 0 . . . 1023. The absolute value of the button status code represents the current status of the mouse button and its status the last time the mouse was read. These status codes are summarized in Table 10-2.

Table 10-2. Button status codes for the mouse.

<i>Status Code*</i>	<i>Description</i>
1	Button is being pressed and was pressed last time.
2	Button is being pressed and was released last time.
3	Button is released and was pressed last time.
4	Button is released and was released last time.

*Absolute value. If the status code is negative, then a key has been pressed.

For example, if the button status code is 3 (or -3), then the mouse button is not being pressed, but it was being pressed the last time that you requested mouse data.

If any key that generates an ASCII code has been pressed on the //c's keyboard, then the button code will be negative. In this situation, you should always read the keyboard (with a PEEK(49152) command) to get the keystroke and then clear the keyboard strobe (with a POKE 49168,0 command). If you don't clear the keyboard strobe, then the button code will remain negative even though no new key has been entered.

A Sample Program

All of the techniques we have described above have been used in the program called MOUSE.DEMO in Table 10-3. The body of this program is a simple loop that continually reads the mouse and displays its current X and Y coordinates and button status code on the screen.

Move the mouse around on your desktop while this program is running and notice how the X and Y coordinates change. As expected, they always range between 0 and 1023.

Table 10-3. MOUSE.DEMO—a program to demonstrate how to read the mouse from an Applesoft program.

```

0  REM "MOUSE.DEMO"
100 D$ = CHR$ (4)
110 KB = 49152: REM KEYBOARD I/O
    LOCATION
120 KS = 49168: REM KEYBOARD ST
    ROBE I/O LOCATION
130 TEXT : PRINT CHR$ (21): HOME
    : POKE 34,3
140 PRINT "APPLE //c MOUSE DEMO
    [Press ESC to end]"
150 PRINT "X VALUE      Y VALU
    E      BUTTON"
160 PRINT D$;"PR#4": PRINT CHR$
    (1): PRINT D$;"PR#0": REM TU
    RN ON MOUSE
170 PRINT D$;"IN#4": REM READ M
    OUSE
180 INPUT "";X,Y,B
190 IF B < 0 THEN KY = PEEK (K
    B): POKE KS,0: IF KY = 155 THEN
    230
210 VTAB 5: PRINT X,Y,B
220 GOTO 180
230 PRINT D$;"IN#0": REM DON'T
    READ MOUSE
240 TEXT : HOME      END

```

You will also notice that the button status code is 4 if you haven't touched the mouse button; referring to Table 10-2, this means "the button is released and it was released the last time, too." If you quickly press the mouse button and then release it (that is, you "click" it), however, the button status will first change to 2, and then 3, before returning to 4 again. If you press the mouse button, keep it down for a while, and then release it (that is, you "press" it), the button status will first change to 2, then to 1 until you release the button, then to 3, and finally back to 4. You can see that these codes make sense by referring to their descriptions in Table 10-2.

The Mouse and Assembly Language

Controlling the mouse from an assembly-language program would be a rather complex chore if it was necessary to write the fundamental I/O drivers

needed to communicate with it. Fortunately, the IIc's firmware contains a set of eight useful subroutines that can be used to control the mouse without the necessity of dealing directly with machine-specific I/O locations. The starting addresses for these subroutines are stored in the form of offsets from the start of page \$C4 of memory in a table beginning at location \$C412 in the IIc's ROM area. Thus, if an entry in this table is \$3D, the starting address for the subroutine defined by that entry is \$C43D. The names for these subroutines, and their positions in the table of offsets is shown in Table 10-4.

Table 10-4. Offset locations for mouse subroutines.

<i>Subroutine Name</i>	<i>Location Where Offset Stored</i>
SETMOUSE	\$C412
SERVEMOUSE	\$C413
READMOUSE	\$C414
CLEARMOUSE	\$C415
POSMOUSE	\$C416
CLAMPMOUSE	\$C417
HOMEMOUSE	\$C418
INITMOUSE	\$C419

The easiest way to call a mouse subroutine is to store its starting address in two consecutive memory locations (low-order byte first, of course) and then use an indirect JMP instruction to pass control to it. For example, if you determine the starting address to be \$C43D, then you can call the subroutine by storing \$3D in location \$300, \$C4 in location \$301, and then executing a "JMP (\$0300)" instruction.

You should make it a practice to always communicate with the mouse by using the built-in mouse subroutines in this way. If you do, then your software will be compatible with the version of the mouse used with earlier members of the Apple II family.

Mouse Screen Hole Locations

The mouse firmware makes use of several main memory screen holes for data storage. As you will recall from Chapter 7, screen holes are memory locations that, although they are located within the IIc's primary text page from \$400...\$7FF, are not used for video display purposes and are not affected by the IIc's built-in video output subroutines. The main memory screen holes used by the mouse firmware are summarized in Table 10-5. The corresponding locations in auxiliary memory are also used by the mouse for the storage of default parameters.

Using the Mouse Subroutines

There are three important points to keep in mind before a mouse subroutine is called:

- 65C02 interrupts must be disabled before calling the subroutine and re-enabled after the subroutine ends. A program that does this will look something like this:

```
PHP      ;Save status
SEI      ;Disable interrupts
.
[call mouse subroutine]
.
PLP      ;Restore status
          (including interrupts)
```

- The mouse X coordinate, MOUXL (\$47C) and MOUXH (\$57C), Y coordinate, MOUYL (\$4FC) and MOUYH (\$5FC), and status byte, MOUSTAT (\$77C), must be moved to data areas within your program before re-enabling interrupts after calling a mouse subroutine. If you don't do this and you try to read mouse data directly from the screen holes after calling a mouse subroutine, the data you read will be wrong if another mouse interrupt occurs before you have done so.
- The 65C02 X and Y registers must be set equal to \$C4 and \$40, respectively ("4" is the mouse port number).

Comparing the //c Mouse with the //e Mouse

Before we take a close look at each of the mouse subroutines, let's outline the important differences between the //c mouse and the mouse that is used with earlier members of the Apple II family (for convenience, we'll call it the "//e mouse").

The //e mouse is interfaced to the Apple //e through an interface card that is inserted into one of its seven peripheral expansion slots. This card contains a special microprocessor that is solely responsible for monitoring the status of the mouse and for sending an active interrupt signal to the IRQ line on the //e's 6502 microprocessor only when a mouse interrupt mode has been selected. When current mouse data is needed, it must be transferred from this interface card into the //e's screen hole locations.

The //c, on the other hand, contains no alternative microprocessor dedicated to the control of the mouse. Mouse control is the responsibility of the same 65C02 that executes all your programs. In order to keep tabs on what the mouse is up to, the //c fixes things up so that a 65C02 IRQ interrupt signal is always generated whenever the mouse is moved. The //c's built-in interrupt-



Table 10-5. Screen holes used by the mouse firmware.

Location Hex (Dec)	Symbolic Name	Description
\$478	MINL	Clamping minimum to set (low)
\$4F8	MINH	Clamping minimum to set (high)
\$578	MAXL	Clamping maximum to set (low)
\$5F8	MAXH	Clamping maximum to set (high)
\$47C	MOUXL	X coordinate (low)
\$4FC	MOUYL	Y coordinate (low)
\$57C	MOUXH	X coordinate (high)
\$5FC	MOUYH	Y coordinate (high)
\$67C	MOUARM	Interrupt "arming" byte [reserved]
\$6FC		Mouse status byte:
\$77C	MOUSTAT	bit 7 : 1 = mouse button down bit 6 : 1 = mouse button down on last read and is still down bit 5 : 1 = mouse has moved since last read bit 4 : [reserved] bit 3 : 1 = VBL interrupt has occurred bit 2 : 1 = button interrupt has occurred bit 1 : 1 = movement interrupt has occurred bit 0 : [reserved]

\$7FC	MOUMODE	Mouse mode byte: bit 7 : [reserved] bit 6 : [reserved] bit 5 : [reserved] bit 4 : [reserved] bit 3 : 1 = VBL interrupts enabled bit 2 : 1 = button interrupts enabled bit 1 : 1 = movement interrupts enabled bit 0 : 1 = mouse on
\$47D	MINXL	X clamping minimum (low)
\$4FD	MINYL	Y clamping minimum (low)
\$57D	MINXH	X clamping minimum (high)
\$5FD	MINYH	Y clamping minimum (high)
\$67D	MAXXL	X clamping maximum (low)
\$6FD	MAXYL	Y clamping maximum (low)
\$77D	MAXXH	X clamping maximum (high)
\$7FD	MAXYH	Y clamping maximum (high)

handling subroutine traps this interrupt before it can get to the user-installed interrupt-handling subroutine, updates the mouse's X and Y coordinates in the main memory screen holes, and changes MOUSTAT (\$77C) to reflect the new status of the mouse. This interrupt is only passed on to the user-installed interrupt-handling subroutine if a mouse movement interrupt mode is active. Mouse button interrupts are generated by enabling VBL interrupts when this mode is selected, polling the button I/O location, RD63 (\$C063), whenever a VBL interrupt occurs, and passing the interrupt through if the button is being pressed.

The key differences between the //e mouse and the //c mouse are as follows:

- The //e mouse never tries to interrupt the 6502 unless a mouse interrupt mode is active; the //c mouse always tries to interrupt the 65C02 when the mouse is moved (assuming that the mouse is on, of course).
- The current data for the //e mouse is stored on the mouse interface card until it is specifically transferred to the main memory screen holes; the current data for the //c mouse is always found in the main memory screen holes.
- The //c mouse will not appear to function in passive mode if 65C02 interrupts have been disabled with a SEI instruction because its movement interrupts will be ignored; the state of the interrupt flag has no effect on the operation of the //e mouse in passive mode.

In the descriptions of the mouse subroutines that follow, you will see references to "mouse position registers." These are the mouse data registers that are located on the //e mouse interface card only; they do not exist on the //c. It is important, however, to pretend that they do exist on the //c so that any software that you do develop will work properly with the //e mouse. For example, the READMOUSE subroutine is used to transfer data from the mouse position registers to the screen hole locations corresponding to the X and Y coordinates and the mouse status location. We know that this really isn't necessary on the //c because those registers don't exist and the current mouse data is already in the right place, ready to be read, but if you choose not to call READMOUSE whenever you want to get mouse data, your program won't run properly with the //e mouse.

The Mouse Subroutines

Let's take a close look at each of the subroutines right now.

SETMOUSE. This subroutine is used to set up the mouse mode. This is done by placing the mouse mode code in the accumulator and then calling this subroutine. The valid modes are as indicated in Table 10-6. On exit, the 65C02 carry flag will be clear if the mode was valid; it will be set if it was invalid.

Table 10-6. Valid mouse mode bytes.

<i>Mouse Mode Byte</i>	<i>Description</i>
\$00	Mouse off
\$01	Passive (transparent) mode
\$03	Movement interrupt mode
\$05	Button interrupt mode
\$07	Movement or button interrupt mode

Note: Add 8 to the mode byte value if vertical blanking interrupts are to be active.

SERVEMOUSE. This subroutine should be called as part of an interrupt-handling subroutine to determine whether the mouse or vertical blanking signal was the source of the interrupt. If either one was responsible, then the 65C02 carry flag will be clear (0); otherwise, it will be set (1). This subroutine also sets up the mouse status location, MOUSTAT (\$77C), so that it can be examined to determine the exact type of mouse interrupt that occurred. A complete description of MOUSTAT can be found in Table 10-5.

READMOUSE. This subroutine must be called to read the mouse position registers and place them in the memory locations used to store the mouse's X and Y coordinates. These memory locations are as follows:

X coordinate : \$47C (low), \$57C (high)
Y coordinate : \$4FC (low), \$5FC (high)

This subroutine also clears bits 1, 2, and 3 of MOUSTAT (\$77C), the interrupt bits, and adjust bits 5, 6, and 7, the movement and button status bits, as necessary.

CLEARMOUSE. This subroutine sets the mouse's X and Y coordinates and position registers to 0. The button and interrupt bits in MOUSTAT (\$77C) are not changed.

POSMOUSE. This subroutine sets the mouse position registers to the same values stored in its X and Y coordinates.

CLAMPMOUSE. This subroutine is used to set the clamping boundaries for the X and Y coordinates. Any mouse position below the clamping minimum will automatically be set to that minimum. Similarly, any position above the clamping maximum will be set to that maximum. Before calling this subroutine, the new clamping limits must be set up as follows:

\$478 (low), \$4F8 (high) for clamping minimum
\$578 (low), \$5F8 (high) for clamping maximum

and the accumulator must be set equal to 0 if the clamping limits for the X coordinate are being set, or to 1 if the clamping limits for Y are being set.

HOMEMOUSE. This subroutine changes the mouse position registers to the coordinates of the upper left corner of the clamping window.

INITMOUSE. This subroutine is normally the one that is called before the mouse is used. It sets up the initial default values for the mouse: a clamping window from \$0 to \$3FF for the X and Y coordinates and an initial position of (0,0).

A Sample Program

The MOUSE.IRQ program in Table 10-7 shows how the mouse subroutines and mouse interrupts can be handled by an assembly-language program. When this program is executed, it first stores the address of a mouse interrupt-handling subroutine at \$3FE/\$3FF, the IRQ user-vector locations. The mouse is then initialized by calling INITMOUSE.

The mouse button/movement interrupt mode is then set by calling SETMOUSE with the appropriate mouse mode code (\$07) in the accumulator. Finally, the mouse coordinates are zeroed by calling CLEARMOUSE and 65C02 interrupts are enabled by executing a CLI instruction.

After this has been done, when the mouse is moved or its button is pressed, the IIc's firmware will pass the interrupt that is generated on through to the IRQHNDL interrupt handler. This subroutine does what all good mouse interrupt handlers should: it calls SERVEMOUSE to see if the mouse caused the interrupt. If the mouse is not responsible, the carry flag is set. If it is a mouse interrupt, then MOUSTAT is examined to see what caused it (a movement or a button press). If it is a movement interrupt (bit 1 of MOUSTAT is 1), then a "M" is displayed on the video screen; if it is a button interrupt (bit 2 of MOUSTAT is 1), then a "B" is displayed.

You should take particular note of the MOUSER subroutine that is called to execute all mouse subroutines that MOUSE.IRQ uses. On entry to MOUSER, the X register must contain the number of the mouse subroutine that is to be called; this number is simply the relative position of the subroutine's offset within the table starting at \$C412 (0 for SETMOUSE, 1 for SERVEMOUSE, and so on). MOUSER gets the proper offset and stores it at MOUSE so that the subroutine can be jumped to with a "JMP (MOUSE)" instruction. It then disables interrupts and calls the mouse subroutine before ending.

The Mouse as a Joystick

Many games available for the IIc require the use of a game controller or a joystick. (A joystick is essentially the same as two game controllers, one that controls the X direction and the other that controls the Y direction.) Since these devices can only be connected to the same interface that the mouse uses,

Table 10-7. MOUSE.IRQ—an example of how to use the mouse subroutines and handle mouse interrupts.

```

1  *****
2  * MOUSE.IRQ *
3  *****
4
5  MOUSTAT EQU $77C      ;Mouse status byte
6
7  MTABLE EQU $C412      ;Start of mouse ROM table
8
9  * Mouse subroutine numbers:
10 SETM EQU 0
11 SERVEM EQU 1
12 READM EQU 2
13 CLEARM EQU 3
14 POSM EQU 4
15 CLAMPM EQU 5
16 HOMEM EQU 6
17 INITM EQU 7
18
19 IRQLOC EQU $3FE        ;User IRQ vector
20
21 COUT EQU $FDED         ;Standard output
22
23 ORG $300
24
25 * Install the interrupt handler:
26
27 LDA #<IRQHNDL
28 STA IRQLOC
29 LDA #>IRQHNDL

```

0300: A9 1D
0302: 8D FE 03
0305: A9 03

(continued)

Table 10-7. MOUSE.IRQ—an example of how to use the mouse subroutines and handle mouse interrupts (continued).

```

0307: 8D FF 03 30 STA IRQLOC+1
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58

030A: A2 07 LDX #INITM
030C: 20 41 03 JSR MOUSER ;Initialize the mouse

030F: A2 00 LDX #SETM
0311: A9 07 LDA #$07 ;(Movement/button interrupt mode)
0313: 20 41 03 JSR MOUSER ;Set the mouse mode

0316: A2 03 LDX #CLEARM
0318: 20 41 03 JSR MOUSER ;Clear mouse position

031B: 58 CLI ;Enable 6502 interrupts
031C: 60 RTS

*****
* This is the interrupt handler *
*****
IRQHNDL PHA
TXA
PHA
LDX #SERVEM
JSR MOUSER ;Service mouse interrupt
BCS INTEXT ;Branch if not mouse interrupt
LDA MOUSTAT ;Get status
LSR
LSR ;Movement bit in carry

```



```

032C: 90 07 59 BCC CHECKBTN ;Branch if no movement
032E: 48 60 PHA
032F: A9 CD 61 LDA #$CD
0331: 20 ED 62 JSR COUT ;Display "M"
0334: 68 63 PLA
0335: 4A 64 CHECKBTN LSR ;Button bit in carry
0336: 90 05 65 BCC INEXIT ;Branch if no press
0338: A9 C2 66 LDA #$C2
033A: 20 ED 67 JSR COUT ;Display "B"
033D: 68 68 INEXIT PLA
033E: AA 69 TAX
033F: 68 70 PLA
0340: 40 71 RTI ;(Don't use RTS!!)
72
73 *****
74 * MOUSE executes the mouse subroutine *
75 * specified by the code in the X *
76 * register. *
77 *****
78 MOUSE PHA
79 LDA MTABLE,X ;Get low byte of subroutine addr
80 STA MOUSE ; and set up for indirect JMP.
81 PLA
82 PHP
83 SEI ;Interrupts off for this!!
84 STX XSAVE
85 STY YSAVE
86 JSR DOMOUSE ;Execute subroutine
87 LDY YSAVE
88 LDX XSAVE
89 PLP
90 RTS
91
0341: 48 78 PHA
0342: BD 12 C4 79 LDA MTABLE,X ;Get low byte of subroutine addr
0345: 8D 65 03 80 STA MOUSE ; and set up for indirect JMP.
0348: 68 81 PLA
0349: 08 82 PHP
034A: 78 83 SEI ;Interrupts off for this!!
034B: 8E 5C 03 84 STX XSAVE
034E: 8C 5D 03 85 STY YSAVE
0351: 20 5E 03 86 JSR DOMOUSE ;Execute subroutine
0354: AC 5D 03 87 LDY YSAVE
0357: AE 5C 03 88 LDX XSAVE
035A: 28 89 PLP
035B: 60 90 RTS
91

```

(continued)



Table 10-7. MOUSE.IRQ—an example of how to use the mouse subroutines and handle mouse interrupts (continued).

035E: A2 C4	92	XSAVE	DS	1	
0360: A0 40	93	YSAVE	DS	1	
0362: 6C 65 03	94				
	95	DOMOUSE	LDX	#\$C4	
	96		LDY	#\$40	
	97		JMP	(MOUSE)	
	98				
0366: C4	99	MOUSE	DS	1	;Subroutine address (low)
	100		DFB	\$C4	;High part is always \$C4)

you would think that you would have to disconnect the mouse and connect the game paddles or joystick before you could use these games.

Fortunately, the //c can be configured in such a way as to “trick” it into thinking that the mouse is a joystick. Once the //c has been so configured, the standard Applesoft paddle reading commands, PDL(0) and PDL(1), or the system monitor paddle reading subroutine, PREAD (\$FB1E), will take their values from the position of the mouse and will not attempt to read the position of the non-existent joystick.

If you want the //c to interpret the mouse as a joystick, then all you need do is turn on the mouse in passive mode. You will recall that this can be done by entering the following series of commands from the keyboard:

```
PR#4 [return] [control-A] [return] PR#0 [return]
```

After this has been done, your game disk can be booted and it should work fine with the mouse. Note, however, that if the software that you are using does not use the Applesoft PDL(0) and PDL(1) commands or the system monitor PREAD subroutine, then this technique will not work.

Mouse I/O Locations

The //c supports several soft switch and status locations that can be used to monitor the status of the mouse and the vertical blanking signal. These are summarized in Table 10-8.

Since the mouse primarily communicates with the //c by generating interrupt signals, it is not surprising that most of the mouse I/O locations have something to do with interrupts. Most of them perform one of the following functions:

- Select when an interrupt is to be generated
- Enable and disable interrupts
- Read the interrupt status
- Clear interrupt conditions
- Read the status of the interrupt enable/disable soft switches

For example, RX0EDGE (\$C05C) and FX0EDGE (\$C05D) can be used to select whether a mouse interrupt is to occur on the rising or falling edge, respectively, of the mouse's X0 signal. The corresponding locations for the Y0 signal are RY0EDGE (\$C05E) and FY0EDGE (\$C05F).

Mouse X0 and Y0 interrupts can be enabled by accessing ENBXY (\$C059) or disabled by accessing DISXY (\$C058). Similarly, vertical blanking interrupts can be enabled by accessing ENVBL (\$C05B) or disabled by accessing DISVBL (\$C05A).

Note that the eight soft switches we have just referred to (from \$C058 . . . \$C05F) can only be used if IOUDISOFF (\$C07F) is first written to.

Table 10-8. Mouse Soft Switch and Status I/O Locations.

<i>Address Hex</i>	<i>(Dec)</i>	<i>Usage</i>	<i>Symbolic Name</i>	<i>Action Taken</i>
\$C015	(49173)	R7	MOUSEXINT	1 = mouse X0 interrupt has occurred 0 = no mouse X0 interrupt
\$C017	(49175)	R7	MOUSEYINT	1 = mouse Y0 interrupt has occurred 0 = no mouse Y0 interrupt
\$C019	(49177)	R7	VBLINT	1 = a VBL interrupt has occurred 0 = no VBL interrupt
\$C040	(49216)	R7	RDXYMSK	1 = mouse interrupts enabled 0 = mouse interrupts disabled
\$C041	(49217)	R7	RDVBLMSK	1 = VBL interrupts enabled 0 = VBL interrupts disabled
\$C042	(49218)	R7	RDX0EDGE	1 = interrupt on falling X0 edge 0 = interrupt on rising X0 edge
\$C043	(49219)	R7	RDY0EDGE	1 = interrupt on falling Y0 edge 0 = interrupt on rising Y0 edge
\$C048	(49224)	R	RSTXY	Clear X0/Y0 mouse interrupt condition

[The action taken for the soft switches from \$C058 ... \$C05F is only taken if access has first been enabled by writing to IOUDISOFF (\$C07F).]

\$C058	(49240)	RW	DISXY	Disable mouse X0/Y0 interrupts
\$C059	(49241)	RW	ENBXY	Enable mouse X0/Y0 interrupts
\$C05A	(49242)	RW	DISVBL	Disable VBL interrupts
\$C05B	(49243)	RW	ENVBL	Enable VBL interrupts
\$C05C	(49244)	RW	RX0EDGE	Interrupt on rising mouse X0
\$C05D	(49245)	RW	FX0EDGE	Interrupt on falling mouse X0
\$C05E	(49246)	RW	RY0EDGE	Interrupt on rising mouse Y0
\$C05F	(49247)	RW	FY0EDGE	Interrupt on falling mouse Y0

\$C063	(49251)	R7	RD63	1 = mouse button is not pressed 0 = mouse button is pressed 1 = mouse has moved to right 1 = mouse has moved up Clear the VBL interrupt condition Disable \$C058-\$C05F IOU access Enable \$C058-\$C05F IOU access
\$C066	(49254)	R7	MOUX1	
\$C067	(49255)	R7	MOUY1	
\$C070	(49264)	R	PTRIG	
\$C07E	(49278)	W	IOUDISON	
\$C07F	(49279)	W	IOUDISOFF	

NOTE: The "Usage" column in the above table indicates how a particular location is to be accessed:

"W" means "write to the location."

"R" means "read from the location."

"RW" means "read from or write to the location."

"R7" means "read and check bit 7 to determine the status."

When an interrupt occurs, it is nice to know what caused it. This can be deduced by examining one of several status I/O locations. If the interrupt was caused by the up/down movement of the mouse, bit 7 of MOUSEYINT (\$C017) will be 1 (that is, if you PEEK this location, the result will be greater than 127). The corresponding location to check for left/right movement is MOUS-EXINT (\$C015). Vertical blanking interrupts can be detected by examining bit 7 of VBLINT (\$C019).

After an interrupt signal is generated, it must be cleared by reading another I/O location. If this is not done, then other interrupt signals will be generated even though no new interrupt has actually occurred. A mouse movement interrupt condition can be cleared by reading RSTXY (\$C048). A vertical blanking interrupt can be cleared by reading PTRIG (\$C070).

You can always determine exactly how the interrupt disable/enable switches have been configured by examining bit 7 of another set of status locations. To check whether mouse movement interrupts are enabled, look at RDXYMSK (\$C040); if bit 7 is 1, then they are. Similarly bit 7 of RDVBLMSK (\$C041) indicates whether vertical blanking interrupts are enabled. RDX0EDGE (\$C042) and RDY0EDGE (\$C043) can be read to determine whether interrupts are to be generated on the falling or rising edge of X0 and Y0, respectively.

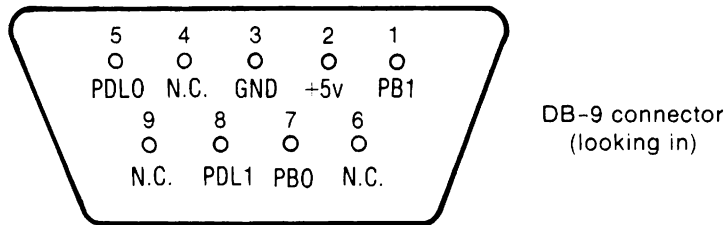
The direction of mouse movement along an axis can be determined by reading bit 7 of locations MOUX1 (\$C066) and MOUY1 (\$C067). If a mouse X0 interrupt has occurred, bit 7 of MOUX1 will be 1 if the mouse has been moved to the right or 0 if it has been moved to the left. MOUY1 can be examined after a Y0 interrupt to determine whether the motion was up or down.

The status of the mouse button can be determined by reading RD63 (\$C063). If bit 7 is 1 then the button is not pressed; otherwise it is.

As you can probably appreciate, writing a program to control the mouse is a fairly complex chore when you are dealing with the mouse at the lowest level by directly monitoring I/O locations. Fortunately, however, you should never need to use any of these I/O locations. Instead, you can use the //c's built-in firmware subroutines and interrupt handler to simplify dialog with the mouse.

The Game Controller Interface

The game controller interface is a very versatile one. As its name suggests, it is primarily used to interface devices that allow you to play video games: devices such as paddles, joysticks, and push buttons. When interfaced to the appropriate supporting circuitry, however, it can also be used to detect light levels, measure the temperature, and perform many other interesting and useful feats.



NOTES: PB = push button (switch) input
 PDL = game controller (paddle) input
 GND = electrical ground
 +5v = +5 volts
 N.C. = not used

Figure 10-1. Pinout diagram for the game connector.

A pinout diagram for the game connector is shown in Figure 10-1. We will be referring to this diagram throughout the remainder of this chapter as we propose several simple interfacing projects that use the signals available through the game connector.

Of the 9 pins on the main game connector, three are not used, two are used for the power supply connections (+5 volts and electrical ground) and four are used for one-bit inputs (2 switch inputs and 2 analog inputs). All of these signals will be discussed in detail in the following sections.

Game Controller Inputs

There are two game controller input pins (also called “paddle” inputs) on the game connector (PDL0 and PDL1), that are normally used to interface two game paddles or one joystick to the //c. These inputs are also often referred to as the analog inputs. Each PDL input is associated with a unique I/O memory location, as shown in Table 10-9. Only bit 7 of these locations is meaningful as we will see shortly.

The game controller inputs are designed to be used with analog devices capable of changing their internal resistances in the range 0-150K ohms in response to a physical phenomenon that is to be measured (such as the position of a game paddle or joystick, the temperature, or air pressure). Such devices are called “transducers” because they are converting a physical phenomenon into an electrical quantity (resistance) that can be quantified by a digital computer like the //c.

Each PDL input is part of an analog-to-digital (A/D) conversion circuit that allows an analog resistance value to be converted (by software) into a digital quantity the //c can handle. The resistor forms part of a simple “RC” (resistor-capacitor) timing circuit that sets the time constant of a special integrated circuit called a NE556 Timer. When this timer is reset, by accessing PTRIG

Table 10-9. Game controller I/O memory locations.

Address		Usage	Symbolic Name	Action Taken (or Status)
Hex	(Dec)			
\$C064	(49252)	R7	PDL0	1 = game controller 0 has not timed out
\$C065	(49253)	R7	PDL1	1 = game controller 1 has not timed out
\$C070	(49264)	R	PTRIG	Reset the game controllers

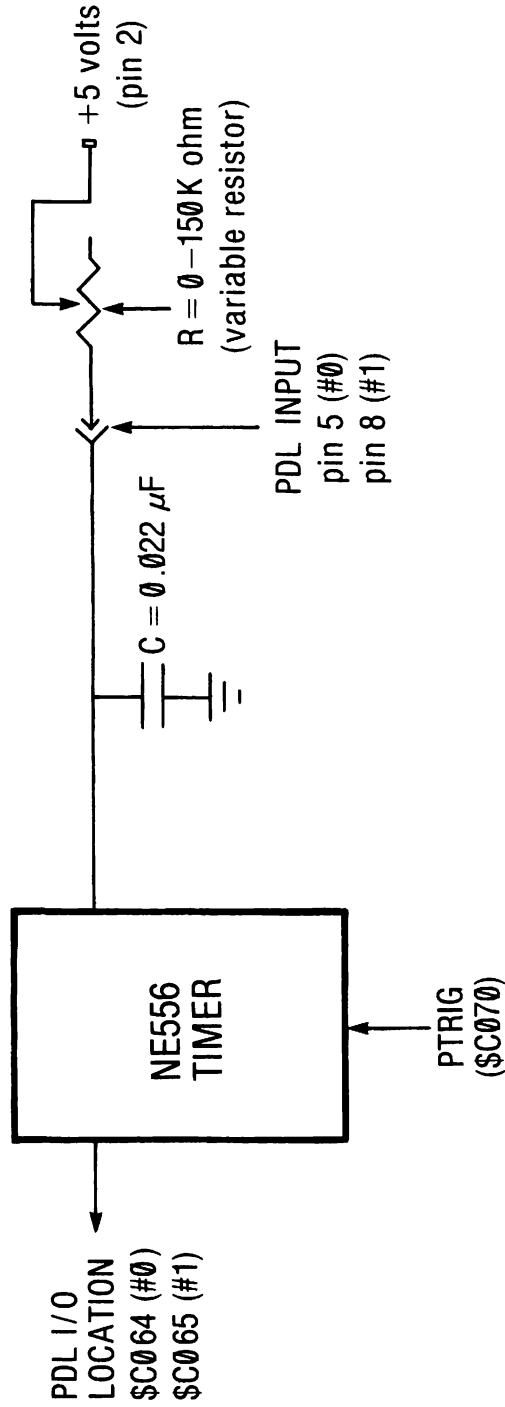
Note: "R7" means "read and check bit 7 to determine the status."

"R7" means "read from the location."

(\$C070), bit 7 of each PDL I/O memory location becomes high (1) but will eventually become low (0) when the timer "times out," that is, the period of time equal to the time constant for both "RC" circuits has elapsed.

To interface a variable-resistor device, all you need to do is connect one of its leads to +5v (pin 2) and the other to one of the PDL input pins. A simplified diagram for one such circuit is presented in Figure 10-2. Since the maximum "R" value recommended by Apple is 150,000 ohms and the "C" value is 0.022 microfarad, the maximum time constant for this circuit is $0.022 \times 150,000$ ohms = 0.0033 second. That is, when the resistance is at its maximum, the time required for the NE556 Timer to bring bit 7 of the PDL I/O memory location low (0) is about 3.3 milliseconds. The time required to do this will change whenever the resistance of the device changes because the RC time constant will also change.

By setting up a program that periodically checks to see whether the NE556 Timer has timed out (by examining bit 7 of the PDL I/O memory location) and increments a counter if it has not, you can easily convert the resistance to a numerical value that varies linearly with resistance. In fact, Applesoft's built-in paddle-reading functions, PDL(0) and PDL(1), do this for you automatically—the counter value they return is an integer between 0 and 255. (You can examine the assembly-language subroutine that these functions use by looking at the PREAD (\$FB1E) subroutine located in the system monitor; it checks for a timeout condition every 11 microseconds.) You should note, however, that the PDL functions assume that your input resistance is in the range 0–150K ohms. This roughly translates to a time constant that ranges from 0 to 2.8 milliseconds and to PDL readings between 0 and 255. (Remember that the PDL subroutine's counter increments every 11 microseconds until the timer has timed out. This means that the maximum allowable time constant is 255×11 microseconds, or 2.8 milliseconds.) If the upper limit of the resistance is higher than 150K ohms, then there will be a "dead area" where the resistance may change but the value calculated stays at 255; if it is lower, then the highest PDL value that can be generated will be less than 255.



NOTE: the RC time constant varies from 0 to 3.3 milliseconds.

Figure 10-2. Block diagram of game controller circuitry.

The PTRIG (\$C070) signal initiates the A/D conversion procedure for both game controller circuits at the same time. Since the NE556 Timer will time out at different times for each game controller (unless their resistances are identical), it is possible that after reading one PDL value that the other game controller is still timing out. If an attempt is made to read this other controller immediately after reading the first one, only the time needed to complete the timing-out process from the first PTRIG will be measured. This leads to a spurious game controller signal that is lower than expected. To avoid this “crosstalk” between paddles, you should wait about 3 milliseconds before reading the other game controller; this delay gives the other game controller a chance to time out. This can be done in Applesoft by placing a short FOR/NEXT loop between the two PDL functions. Here is an example of how to do this:

```
100 X=PDL(0):FOR I=1 TO 10:NEXT: Y=PDL(1)
```

Two devices that are commonly interfaced to the game controller inputs are the game paddle and the joystick. A game paddle is a device that controls the signal at one PDL input only; it typically takes the form of a knob that you can rotate with your hand. As the knob is rotated, the resistance value changes linearly. A joystick allows you to control both PDL inputs at once in such a way that the two-dimensional position of the joystick can be easily detected by reading the two game controller values.

There is no reason to restrict the game controller inputs to use with game paddles and joysticks, however. Any device that provides a fluctuating resistance value within the 0–150K range could also be interfaced and its resistance converted to a value between 0 and 255 using the Applesoft PDL() commands or their assembly-language equivalents.

Examples of two such useful devices are a thermistor and a photoresistor. A thermistor is a device that changes resistance with temperature. Several types of thermistors are available, including types that will generate resistances within the 0–150K ohm range for most temperatures that you would want to measure. Unfortunately, most thermistors are not sensitive to small temperature changes, such as those that might occur in a home, so the range of PDL values read may not be large. In addition, the values generated may not vary linearly with temperature. Nevertheless, you can calibrate the thermistor by preparing a table of actual temperatures (measured with a standard thermometer) and their associated paddle readings. This will at least allow you to estimate the temperature from a given “paddle” reading.

A photoresistor is a device that changes resistance with the amount of light shining on it. The greater the light intensity, the lower the resistance. You would calibrate this device by preparing a table of light intensities (as measured by a light meter) and their associated “paddle” readings.

Let’s hook up a photoresistor to the game connector to show you how it works. A handy photoresistor to use is a cadmium sulfide one that is readily available from Radio Shack (part number 276-116). All you have to do to

interface it to a game controller input, say PDL0, is to connect one leg of the photoresistor to +5 volts (pin 2) and the other leg to PDL0 (pin 5). Once you have done this, you can read its current setting by using the Applesoft PDL(0) command. Enter the following program and then run it:

```
100 PRINT PDL (0)
200 GOTO 100
```

While the program is running, turn off the room lights to verify that the “paddle” value increases when there is less light. If you have a dimmer light switch, slowly turn the light intensity up and see how the value slowly decreases until it goes to 0 in very bright light.

Push Button Inputs

There are two one-bit input ports on the game connector that are normally used to read the state of external switches connected to them. These are the so-called “push-button” input ports. These ports, and the switches themselves, are usually referred to by their descriptive names: PB0 and PB1 (or sometimes SW0 and SW1).

The //c assigns one I/O memory location to each of the push-button input ports, but only bit 7 at that location is actually used. These locations are shown in Table 10-10. By reading the memory location for a particular push-button input (using an Applesoft PEEK or an assembler LDA) and examining bit 7, you can determine whether a switch is being pressed or not. By convention, if the bit is set to 1, then the switch is considered to be on (that is, pressed); if it is cleared to 0, the switch is considered to be off (that is, released). You should note, however, that it is possible for a switch to be connected in such a way that exactly the opposite result is observed. More on this later.

A switch is a simple electrical component. It is typically used to allow you to complete an electrical circuit between its two contacts in order to turn something on and to break this circuit in order to turn something off. (Some

Table 10-10. Push button I/O memory locations.

<i>Address</i>		<i>Usage</i>	<i>Symbolic Name</i>	<i>Action Taken (or Status)</i>
<i>Hex</i>	<i>(Dec)</i>			
\$C061	(49249)	R7	PB0	1 = push button 0 or OPEN-APPLE is pressed
\$C062	(49250)	R7	PB1	1 = push button 1 or SOLID-APPLE is pressed

Note: “R7” means “read and check bit 7 to determine the status.”

switches can have more than two contacts, but we'll ignore them for the moment.) There are many varieties of switches, but the variety that is commonly connected to the push button inputs is, you guessed it, the push button. This is because they are ideally suited as triggers for such video game weaponry as laser cannons, machine guns, and so on.

Switches can be classified into one of two categories: "momentary contact" or "fixed contact." A momentary-contact switch is one that returns to its initial "resting" position immediately after you take your finger off it. All of the keys on the //c's keyboard (except the CAPS LOCK key) are examples of such a switch.

A fixed-contact switch is one that can be turned on or off and that will stay on or off, as the case may be, after you have taken your finger off it. Examples of fixed-contact switches are the CAPS LOCK key on the //c's keyboard, a standard light switch, and a toggle switch.

Two other special terms are used to describe the operation of momentary-contact switches: "normally open" and "normally closed." A switch is said to be normally open if, when it is not being pressed, no connection is made between its contacts. Conversely, a normally closed switch is one in which the contacts are closed when it is not pressed.

It is important to know whether the momentary-contact switch that you wish to interface to the game connector is normally open or normally closed, because the interface circuit that you must build will be different for each type of switch. Figure 10-3 sets out the two alternative circuits. These circuits have been designed in such a way that if the switch is not being pressed, then the input to the push button pin is grounded and when it is being pressed, it is connected to 5 volts. This ensures compatibility with Apple's on/off push-button convention referred to earlier.

It is easy to install your favorite type of switch, be it momentary contact or fixed contact, normally open or normally closed, to the game connector. You must install it, however, when the power to the //c is off! Let's assume you have a normally open push button switch and you want to install it as PB1. Following Figure 10-3 (a), connect a wire from one switch contact to the +5v line (pin 2 on the game connector), another wire from the other contact to PB1 (pin 1), and then connect a 1,000-ohm resistor between PB1 (pin 1) and ground (pin 3). (This resistor ensures that the input to the connector will not "float" between 1 and 0 when the switch is not pressed and will also prevent a short-circuit when the switch is pressed.)

You can easily determine whether or not a push button is being pressed by examining bit 7 of the I/O memory location that the //c reserves for that button. As explained earlier, if this bit is on (1), then the button is being pressed; if it is off (0), the button is not being pressed. This means that if you PEEK this memory location from Applesoft, then the number you read is greater than or equal to 128 if the button is pressed or less than 128 if it is not.

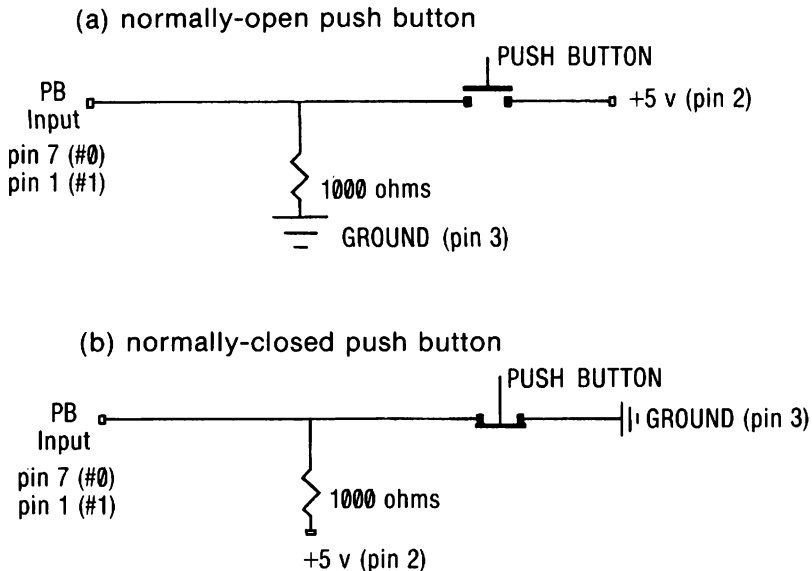


Figure 10-3. Interfacing push buttons to the game

Two keys on the //c's keyboard are actually directly connected to the game connector's push-button input lines. These are the OPEN-APPLE and SOLID-APPLE keys that flank the space bar. These two keys are connected to PB0 and PB1, respectively.

The presence of these two keys enables you to easily experiment with the concept of game-paddle switches without having to do any circuit design at all. Let's write a simple little program to test the status of PB0, the OPEN-APPLE key.

The I/O memory location reserved for PB0 is 49249. To read this location from an Applesoft program, you would use the PEEK(49249) command. Enter the following simple Applesoft program and run it:

```
100 IF PEEK(49249)>127 THEN PRINT "DOWN WE GO!"
200 IF PEEK(49249)<128 THEN PRINT "
    BACK UP AGAIN!"
300 GOTO 100
```

While the program is running, periodically press and release the OPEN-APPLE key. You will find that when it is pressed, the message

DOWN WE GO!

will appear, and that when it is released, you will see the message

BACK UP AGAIN!

By changing the address that is PEEKed, you can easily test the status of any of the other push buttons, including the one you wired up yourself.

Remember that the switches connected to the push-button inputs on the game connector need not be push buttons. Any type of switch can be connected, including toggle switches, reed switches, blow switches, pressure switches, and magnetic switches.

Further Reading for Chapter 10

Note: All of the game connector experiments referred to in the following articles use the 16-pin game connector that is found on the motherboard of the Apple //e, the Apple II Plus, and the Apple II. This connector does not exist on the //c. However, the connector signals used in the experiments are available on the //c's mouse/game connector.

On paddle input in general . . .

P. Baum, "Nibbling at the Game Paddle Port", *Nibble*, October 1984, pp. 100–105. A comparison of the game paddle circuitry in the Apple //c and earlier models.

On reading the game paddles . . .

B. Sander-Cederlof, "Reading Two Paddles At the Same Time", *Apple Assembly Line*, March 1982, p.1. A program to simultaneously read two game paddle inputs.

On interfacing a lie detector . . .

D.B. Curtis, "To Tell the Truth", *Kilobaud Microcomputing*, August 1981, pp. 87–89. How to hook up a lie-detecting device to the game paddle inputs.

On interfacing a joystick . . .

"Dual Joysticks for Under \$15.00", *Nibble*, Vol. 1, No. 2 (1980), p. 13. How to hook up a joystick to the game paddle inputs.

On interfacing a thermistor . . .

C.J. Kershner, "A Digital Thermometer for the Apple II", *Micro*, March 1980, p.21. How to hook up a thermistor to the game paddle inputs.

On interfacing a light pen . . .

D.J. Lilja, "Build a Simple Light Pen for the Apple II", *Byte*, June 1983, pp. 395–406. How to hook up a light pen to the push button inputs.

On interfacing a weather map receiver . . .

K.H. Sueker, "Apple FAX: Weather Maps on a Video Screen", *Byte*, June 1984, pp. 146–151. This fascinating article describes how you can receive

broadcasted weather maps by connecting some circuitry to a push button input.

On TTL logic and digital electronics in general . . .

D. Lancaster, TTL Cookbook, Howard W. Sams and Co., Inc., 1976.

11

The Serial Interface Ports

Not counting the disk drive, the two most common peripherals attached to a microcomputer seem to be a printer and a modem. The reasons for the popularity of printers are obvious, so we won't touch on them here.

Modems are used to communicate with other computers over standard telephone lines. These computers can be large mainframes that contain enormous databases or simply other personal computers like the //c. Modems are becoming increasingly popular as more people begin to appreciate the convenience of being able to interactively tap the information stored on a remote computer, information that would be difficult to locate in any other way.

The //c has two built-in interfaces, called serial ports, that allow devices such as printers and modems to be easily connected to it. You can take a look at these ports by turning the //c around so that its back panel is facing you. Serial port 1 is the special 5-pin connector (called a DIN-5 connector) near the far right with the small drawing of a printer just above it. Serial port 2 is on the left side, next to the mouse/game connector, and it has a drawing of a telephone handset above it.

In this chapter we will be examining in detail how to make use of these serial ports. Included will be discussions of serial interfaces in general, the two "6551" integrated circuits that simplify the data transmission process, and the built-in firmware that can be used to control how data is sent to and read from serial devices such as printers and modems.

Serial Transmission of Data

There are two main methods that are used to transfer data from a computer to an external device: the parallel method and the serial method.

When the parallel method is used, each bit of a byte is simultaneously transmitted to the device along eight wires (one for each bit). This method typically uses a few extra control (or "handshaking") lines: a "busy" line that the receiver can use to indicate that it is not yet ready to receive more data and a "ready" line that the transmitter can use to notify the receiver that data is ready to be sent.

The serial method is the one we are more interested in because it is the one used by the //c. When this method is used, a byte is decomposed into a series of bits and transmitted down one wire only; logical “1”s and “0”s are differentiated by using a different voltage level for each. Handshaking lines similar to those used in a parallel transfer are also often used to control various aspects of the communications link.

The RS-232-C Standard

There are probably an infinite number of ways that two serial devices can be physically connected to permit a useful transfer of information to take place. Even so, it is very desirable that a standard method be used so that any serial device can communicate with any other, even if the devices are manufactured by different companies.

The Electronics Industries Association published its now famous RS-232-C standard in 1969 in an attempt to define a common hardware protocol to be used for serial data communications. (“RS-232-C” stands for “recommended standard number 232, revision C.”) This standard defines the functions of the electrical signals that are permitted to be sent from one serial device to another, the voltage levels of these signals, and even the physical connectors that must be used. The //c does not rigidly adhere to the RS-232-C standard, but the differences will not normally prevent you from communicating with most serial devices that do.

Data Communications Protocols for Serial Communications

Even if the hardware link between two serial devices has been properly set up, the devices will still not be able to understand each other unless they both use the same protocol for exchange of data.

Such a protocol will first define a transmission speed which is to be used for sending and receiving the serial bit stream. The speed is called the baud rate and, for most purposes, is simply equal to the number of bits transmitted per second. Common baud rates for modems are 110, 300, and 1200. Serial printers like the Apple Imagewriter operate at 9600 baud.

The protocol will also define the following key factors:

- The character encoding scheme (ASCII, EBCDIC, Baudot)
- The method used to synchronize the data flow
- The number of data bits
- The order in which data bits are transmitted
- The error-checking method used (parity, checksum)

The protocol that is usually used on microcomputers to transmit data to another serial device is the one shown in Figure 11-1. This method involves the transmission (for each byte to be sent) of a stream of bits in the following order (common values are given in parentheses):

- One start bit
- The data bits (5, 6, 7, or 8)
- An optional parity bit (even, odd, mark, space)
- The stop bit or bits (1, 1.5, 2)

The number of data bits, the parity, and the number of stop bits are said to define the “data format” of the transmission. The serial receiver and transmitter must be using the same data format in order for a successful communications link to be established.

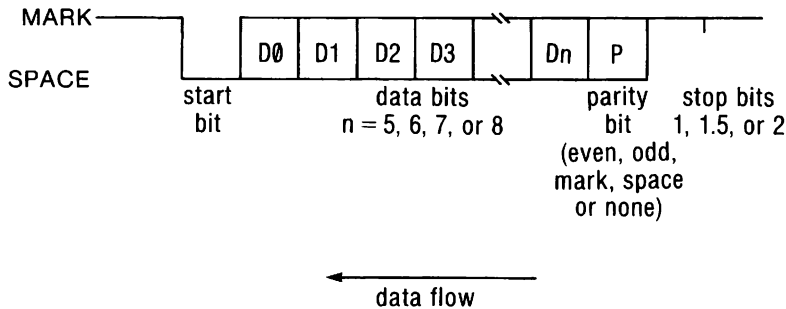


Figure 11-1. The asynchronous serial character transmission protocol.

Let's take a closer look at this protocol right now.

Start Bit

The transmit line is normally kept at a “mark” level (logic “1”) until a character is ready to be sent. A “start bit” is then sent by changing the transmit line to a “space” level (logic “0”) for one bit time; this start bit acts as a signal to the receiver that the data bits for one character are about to follow. When an asynchronous transmission method (such as the one we are discussing) is being used, the receiver must be able to recognize when each character being sent and it is the use of a start bit that allows it to do so.

Data Bits

After the start bit has been sent, the actual data bits are transmitted, one by one, at the rate dictated by the baud rate. Bit 0 of the byte being transmitted

is sent first, then bit 1, bit 2, and so on. The number of data bits which are sent will depend on the particular data format being used by the two serial devices; it normally ranges from 5 to 8 bits. When ASCII codes are being transmitted, 7 or 8 bits are sent; if binary data bytes are being sent, 8 bits must be sent.

Parity Bit

Once all the data bits have been sent, a parity bit will be sent if the data format being used requires it. There are four different types of parity schemes that can be used:

- Mark parity (always 1)
- Space parity (always 0)
- Even parity
- Odd parity

If mark or space parity is being used then the parity bit is always fixed to 1 or 0, respectively. If even parity is used, then the parity bit is adjusted so that the total number of "1"s in the data bits and the parity bit is an even number. When odd parity is used, the adjustment is made so that the total number of "1"s is odd. For example, if the data bits being sent are represented by "10100011", and you are using odd parity, the parity bit would be set to 1; this provides an odd total of "1"s (five).

A parity bit is inserted into the bit stream to permit the receiver of the data to determine if a transmission error occurred. If the receiver is using the same data format as the transmitter and an incorrect parity bit is received, then the bit stream must have been inadvertently garbled. Such errors invariably arise from noise and electrical interference. The parity method of checking for errors is certainly not foolproof, however, since multiple bit errors could easily cancel each other out. Far more elaborate methods must be used if you want to ensure that the data received is, indeed, the same as the data transmitted.

Stop Bits

To indicate the end of the transmission of the data byte, the transmitter sends one or more stop bits to the receiver. These stop bits are equivalent to "1" bits in that the transmitter signal is kept in the mark state. The number of stop bits sent will depend on the data format agreed upon between transmitter and receiver. The most common values are 1, 1.5, and 2. After the stop bits have been sent, the transmitter will stay in the marking state until the next data byte is ready to be sent.

Data Transmission Errors

There are three general classes of errors that a receiver of data can easily detect:

- Framing errors
- Overrun errors
- Parity errors

Framing errors occur when the receiver fails to detect a stop bit when it expects one. This type of error usually occurs when the receiver and transmitter are not using the same data format or baud rate.

Overrun errors occur when a data byte that is received by a serial device is not read by the microcomputer that controls that device before another byte arrives.

Parity errors occur if the parity bit that is received is not consistent with the data bits that have been read. If parity errors consistently occur, then the transmitter and receiver are probably using different baud rates or data formats. If they occur occasionally, then they are probably due to transmission noise that has scrambled the bit stream.

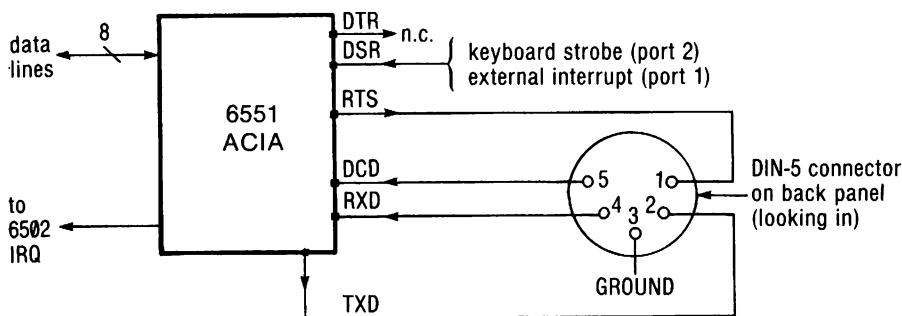
The 6551 ACIA

Each serial port on the //c is controlled by a complex integrated circuit called a 6551 ACIA (Asynchronous Communications Interface Adapter). The main functions that the 6551 performs are as follows:

- Parallel to serial conversion of outgoing data
- Serial to parallel conversion of incoming data
- Handshaking control
- Interrupt handling

When you want to send a byte out the serial interface, all you need do is present it to the 6551. The 6551 automatically converts the parallel data (the byte) into a serial bit stream and frames it by adding the start bit, the proper number of stop bits, and the proper parity bit. The 6551 also takes care of transmitting the bits at the proper baud rate.

A block diagram of the 6551 is shown in Figure 11-2. This diagram shows the correspondence between the various I/O lines on the 6551 and the pins on the DIN-5 port connector on the back panel of the //c. Three-character mnemonics are used to refer to the transmit, receive, and handshaking lines used by the 6551. These mnemonics come from the RS-232-C standard and have the following meanings:



Recommended pin connections to RS-232-C equipment:

FROM DIN-5 signal	TO	
	Modem (DCE)	Printer (DTE)
RTS (pin 1)	DTR (20)	DSR (6)
TXD (pin 2)	TXD (2)	RXD (3)
GND (pin 3)	GND (7)	GND (7)
RXD (pin 4)	RXD (3)	TXD (2)
DCD (pin 5)	DSR (6)	DTR (20)

NOTE: The pin numbers given for DCE and DTE refer to pins on the DB-25 connector defined by the RS-232-C standard.

Figure 11-2. Block diagram of the 6551 ACIA.

TXD—Transmit Data
 RXD—Receive Data
 RTS—Request to Send
 CTS—Clear to Send
 DTR—Data Terminal Ready
 DSR—Data Set Ready
 DCD—Data Carrier Detect

The //c is a member of a class of devices that the RS-232-C standard calls data terminal equipment (DTE). DTE devices are usually the primary sources or destinations of data in a communications link; common DTE devices are terminals and printers. The other class of devices defined by the RS-232-C standard is called data communications equipment (DCE). DCE devices are usually data-link intermediaries that are responsible for maintaining a connection and passing data between two DTE devices. A modem is the standard example of a DCE device.

The RS-232-C signals on both DTE and DCE are labeled from the point of view of DTE only. For example, a DCE device does not transmit data on its TXD line; rather, data is transmitted to it (from a DTE device) on that line. Similarly, a DCE device does not receive data on the RXD line; in fact, the DTE device receives data from the DCE device on that line.

DTE devices are usually connected to DCE devices by an electrical cable that simply connects each line on the DTE to the corresponding line on the DCE. However, when two DTE devices are being connected (such as the //c and a serial printer), you can't do this because both devices would be sending information on the very same lines. To enable two DTEs (or two DCEs) to communicate properly, you have to cross some wires so that one device's output lines are connected to the other's input lines. Here is an example of one wire-crossing scheme that is commonly used:

TXD → RXD
RXD → TXD
RTS → CTS
CTS → RTS
DSR → DTR
DTR → DSR
DCD → DCD (unaffected)

This crossing of wires can either be done inside the RS-232-C cable itself (Apple has done this with its Imagewriter printer cable) or by attaching a special device called a “modem eliminator” or a “null modem” to a standard cable.

Let's take a look at each of the RS-232-C signals used by the 6551 right now and see how they are used to communicate with modems (or any other DCE device) and printers (or any other DTE device).

TXD. The TXD signal is a 6551 output. It is connected to a modem's TXD line or a printer's RXD line. The serial bit stream is sent down this line to the remote device.

RXD. The RXD signal is a 6551 input. It is connected to a modem's RXD line or a printer's TXD line. The 6551 monitors this line in order to read the incoming serial bit stream.

RTS. The RTS signal is a 6551 output. It is usually connected to a modem's RTS line or a printer's CTS line, but on the //c it is meant to be connected to the external device's DTR (modem) or DSR (printer) line instead. The remote device monitors the status of this line and will not send data to the 6551 until it is in a low (0) state.

CTS. The CTS signal is a 6551 input. It is supposed to be a signal originating from a modem's CTS line or a printer's RTS line to indicate that the external device is ready to receive data. However, on the //c, the CTS line is not connected to the DIN-5 plug and is always kept in a low voltage (0) state.

DTR. The DTR signal is a 6551 output. It is usually connected to a modem's DTR line or a printer's DSR line to indicate that the power to the 6551 is on and that it is functioning properly. On the //c, however, the DTR signal is not used.

DSR. The DSR signal is a 6551 input. It is usually connected to a modem's DSR line or a printer's DTR line so that the 6551 can detect whether the remote device is powered up and ready to receive data. On the //c, however, the DSR input on serial port 2 is connected to the //c's keyboard strobe signal and, on serial port 1, to one of the input lines on the external disk drive connector. These connections have been made to allow a 6551 interrupt signal to be generated by the keyboard or by a device connected to the disk drive connector (although the add-on disk drive for the //c currently does not use this signal). We will be discussing 6551 interrupts at the end of this chapter.

DCD. The DCD signal is a 6551 input. It is primarily used with modems to indicate whether a proper telephone connection has been established with the remote computer. If it has, then the DCD input signal will be low (0). For the //c, however, Apple has recommended that the DCD input come from a modem's DSR line or a printer's DTR line. If this is done, then the state of DCD simply reflects whether the remote device is powered up and ready to receive data and does not relate to the state of the telephone connection at all.

The 6551 contains four 8-bit registers that are mapped to locations in the //c's I/O memory space. These registers are used to control various aspects of the communications link to devices that are attached to the serial ports. Let's look at each of them now and see how they are used.

6551 Control Register

The control register is a write-only register that is used to set the following communications parameters:

- The baud rate
- The number of data bits (word length)
- The number of stop bits
- Whether an internal clock or external baud rate clock is to be used (The //c does not support the external clock option.)

The address of the port 1 control register is \$C09B; the port 2 control register is found at \$C0AB. The meanings of each of the bits in the control register are shown in Figure 11-3.

For example, to set up port 1 for a baud rate of 1200 baud, a word length of 8 bits, and one stop bit, you would execute the following instructions:

```
LDA #$18  
STA $C09B
```

This sets up a bit pattern of 00011000 in the control register. You can see by examining Figure 11-3 that this is the pattern required for this configuration.

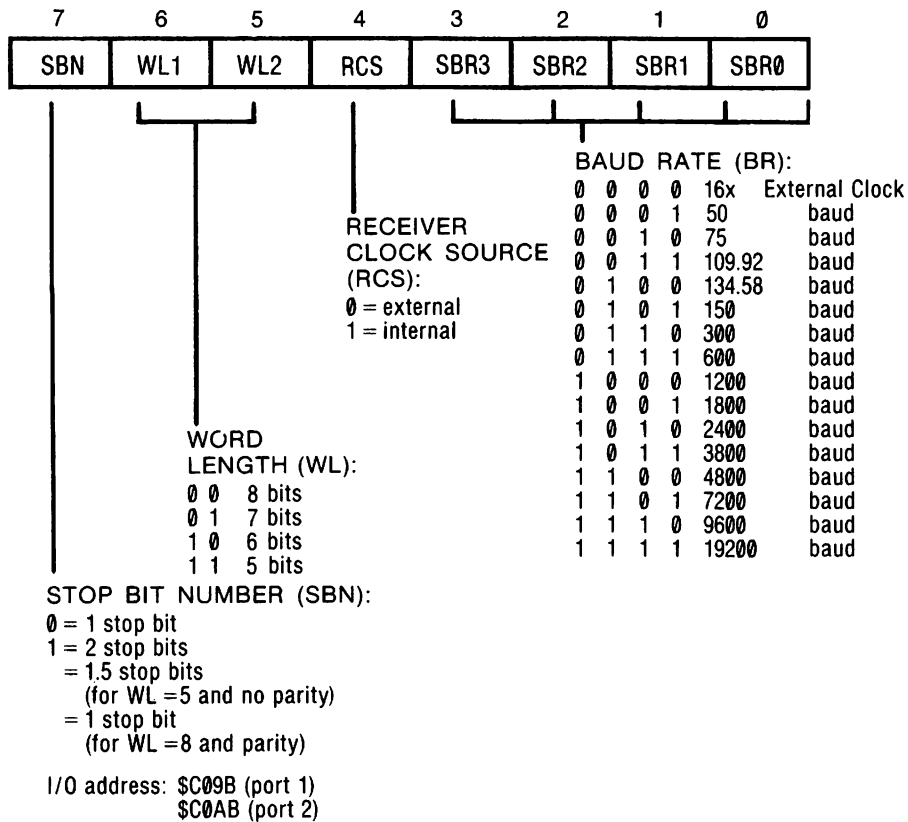


Figure 11-3. The 6551 control register.

6551 Command Register

The command register can be used to control the operating modes of the 6551 as well as the parity bit of the data being transmitted. It is located at \$C09A for port 1 or \$C0AA for port 2. The meaning of each bit in the command register is summarized in Figure 11-4.

You can control the state of two handshaking lines on the serial interface using the 6551 command register: DTR and RTS. As we saw in the previous section, the DTR signal is not connected in the //c implementation of the 6551; nevertheless, it must be set low before the 6551 will operate properly. This can be done by storing a 1 in bit 0 of the 6551 command register. The RTS signal must also be set low in normal operating modes; if it is set high, the remote device will not send any data to you.

The command register can also be used to enable or disable transmitter and receiver interrupts. For example, if bit 1 is 0, then whenever the 6551's receiver register is full (that is, data has arrived and is ready to be read), a

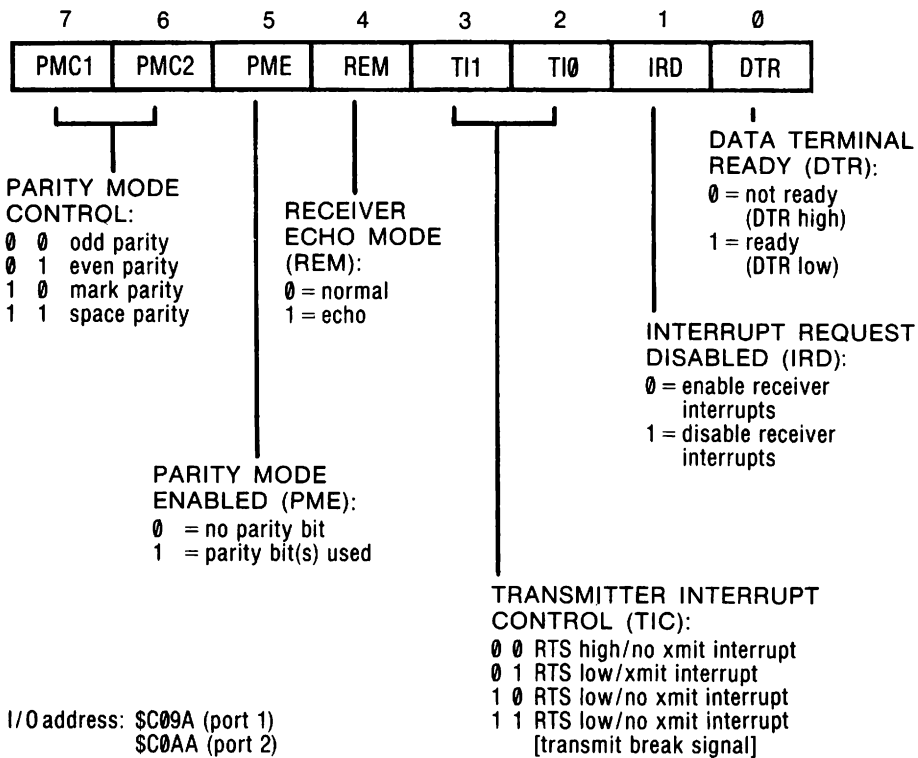


Figure 11-4. The 6551 command register.

65C02 IRQ interrupt will be generated. A similar interrupt will be generated when the transmitter register is empty (that is, the 6551 is ready to send data) if bits 3 and 2 are set to 0 and 1, respectively. We will be discussing 6551 interrupts in greater detail at the end of this chapter.

6551 Status Register

The 6551 status register is located at \$C099 (port 1) and \$C0A9 (port 2). This register can be examined to determine the status of various 6551 functions and to detect whether receiver errors have occurred. The meaning of each bit in the status register is summarized in Figure 11-5.

The status register is most often used to determine when it is possible to read incoming data or send outgoing data. For example, if you want to send data out the serial port, you would wait until the transmitter data register is empty (it's empty when bit 4 of the status register is 1) and then store the data in the data register (see below). Similarly, you can read incoming data by waiting until the receiver data register is full (it's full when bit 3 of the status register is 1) and then reading the data from the data register.

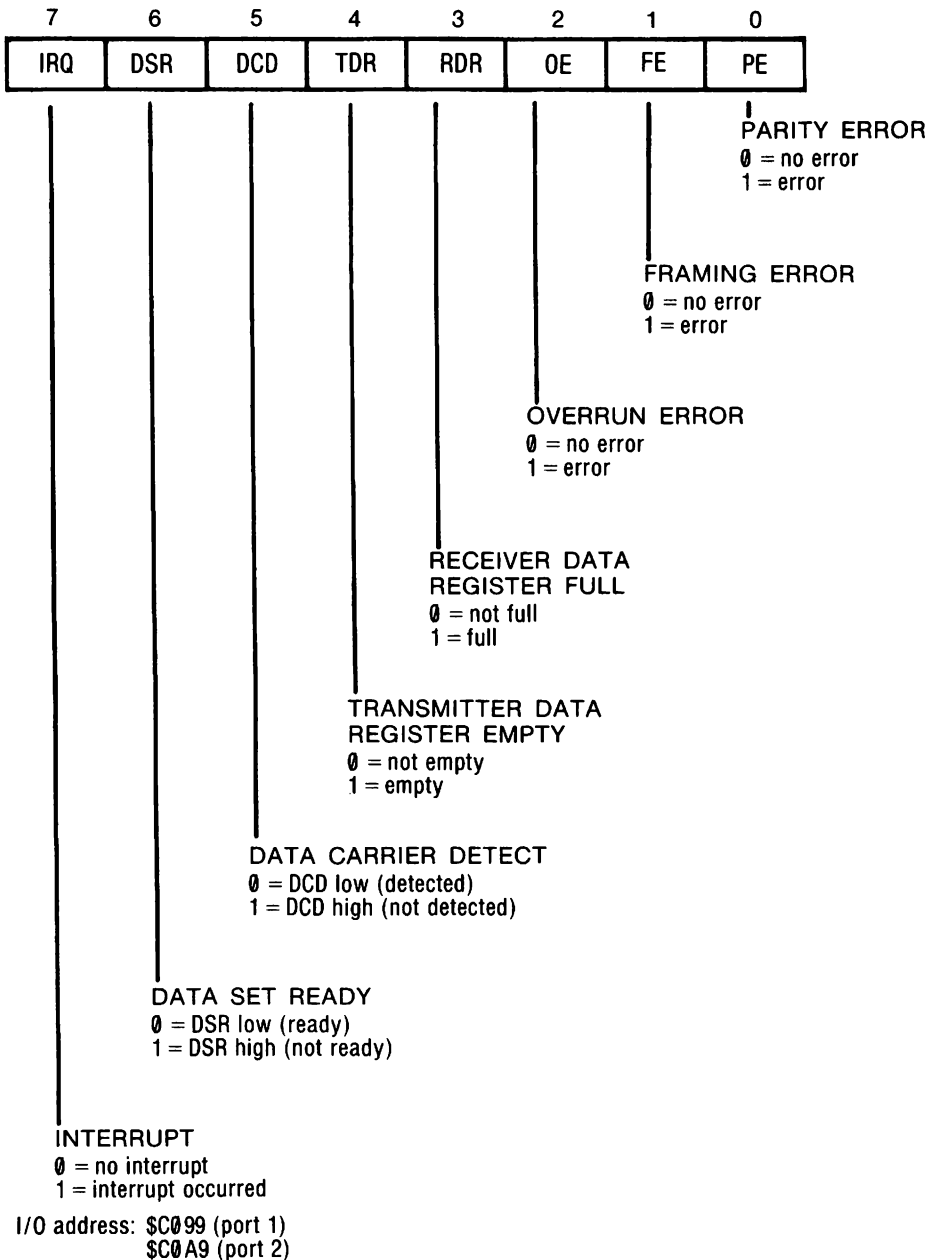


Figure 11-5. The 6551 status register.

The status register can also be used to monitor the states of two incoming handshaking lines, DCD and DSR, which were described earlier. Briefly, the state of the DCD line can be tested to determine whether the remote device

is ready to receive data; if it is, then DCD will be low (that is, bit 5 of the status register will be 0). The DSR bit can be examined to determine the state of the keyboard strobe (port 2 only) or the external interrupt line (port 1 only). We'll be examining these signals in detail at the end of the chapter.

You can also check the status register to determine whether a 6551 interrupt condition has occurred. When an interrupt occurs, bit 7 of the status register will be 1.

6551 Data Register

The data register is located at \$C098 (port 1) and \$C0A8 (port 2). This is where you store bytes that you want to send out the serial port and also where you read incoming bytes.

The data register should only be read when the receiver data register is full and only be written to when the transmitter data register is empty. The states of the receiver and transmitter data registers can be determined by examining the relevant bits in the 6551 status register.

Configuring the Serial Ports

The two external serial ports on the //c are identical. This means that, in principle, it doesn't really matter which port an external serial device is connected to. However, you have a choice of two built-in firmware subroutines to control usage of each port. These subroutines are used to configure a serial port as either a printer port or a communications port.

When you are communicating with a device like a printer that can receive, but not transmit, data, then the serial port to which it is connected should be configured as a printer port. When this is done, several special commands become available that can be used to adjust parameters that affect the data sent to a printer.

On the other hand, if you are communicating with a device that can both receive and transmit data, then the serial port should be configured as a communications port. Such devices include other personal computers, modems, and terminals.

When the //c is first turned on, serial port 1 is automatically configured as a printer port and serial port 2 as a communications port. Although it is possible to change this configuration (keep reading to find out how), it is a good idea to always connect a printer to port 1 and a modem (or other two-way communications device) to port 2.

We're now going to take a closer look at the characteristics of a printer port and a communications port. In the examples which follow, we will assume, for convenience, that port 1 is the printer port and port 2 is the communications port.

Characteristics of a Printer Port

When the //c is first turned on, it automatically configures serial port 1 as a printer port having the following characteristics:

- A baud rate of 9600
- A data format of 8 data bits, no parity, 2 stop bits
- An 80-column line width (This means that a carriage return character is automatically printed whenever 80 characters are printed without an intervening carriage return.)
- A line feed character is sent after every carriage return character

If you don't like these parameters, you can select different ones by using special printer commands that we'll be looking at shortly.

All you have to do to direct character output to a port 1 printer is to enter a PR#1 command from the keyboard or to execute the following Applesoft statement:

```
PRINT CHR$(4);"PR#1"
```

from within an Applesoft program. You can use the PR#0 or PR#3 command after everything has been printed to direct output to the video display once again.

Once printer port 1 has been turned on with a PR#1 command, there are several commands that can be sent to the printer port firmware to modify the printer port characteristics. These commands, and the functions they perform, are summarized in Table 11-1. These commands can be entered by typing them in from the keyboard or by using the Applesoft PRINT command to send them directly to the printer port.

To select a printer command, you must first enter a special command prefix character, followed by the command itself. The default command prefix character is [control-I], but this can be changed to another control character by entering the current prefix character followed by the new one. You will want to do this in situations where it is necessary to send the prefix character itself to the printer without it being eaten by the firmware.

For example, if the current command character is [control-I] and you want to change it to [control-K], you would type (or print) the two characters

```
[control-I] [control-K]
```

Note that you must not change the prefix character to [control-A], [control-C], [control-H], [control-J], [control-L], [control-M], or [control-Y]. These control characters are used for special purposes by the //c's firmware.

Let's look at a couple of examples of how to enter printer commands. Suppose your printer operates at 1200 baud and that it automatically advances the paper after it receives a carriage return code. You will not be able to

Table 11-1. Printer port commands.

Command

Description

nnn

Set the new line width to nnn (1 . . . 255). Follow this command with [return] or N.

nnB

Set the baud rate to the value corresponding to the nn code:

nn

Baud Rate

nn

Baud Rate

1

50

9

1800

2

75

10

2400

3

110

11

3600

4

135

12

4800

5

150

13

7200

6

300

14

9600

7

600

15

19200

8

1200

nD

Set the data format to the value corresponding to the n code:

n

Data Bits

Stop Bits

0

8

1

1

7

1

2

6

1

3

5

1

4

8

2

5

7

2

6

6

2

7

5

2

I

Send printer output to the video screen as well.

K

Don't automatically send a line feed character after every carriage return character. If your printer is double spacing all output, then you must enter this command.

L

Automatically send a line feed character after every carriage return character. If your printer is not advancing the paper after the printing of each line, then you must enter this command.

nnnN

Set the new line width to nnn (1 . . . 255) and don't send printer output to the video screen.

nP

Set the parity to the state corresponding to the n code:

n

Parity

0,2,4,6

None

1

Odd

(continued)

(continued)

Table 11-1. Printer port commands (continued).

Command	Description
	3 Even
	5 Mark (1)
	7 Space (0)
R	Reset the 6551 ACIA and turn off the printer port.
S	Send a 233 millisecond break signal to the printer. This signal is used to synchronize some printers with the serial output stream.
Z	Ignore further command characters until the next PR#1 or [control-RESET]. Disable the automatic insertion of carriage return characters.

communicate with this printer right after the //c is turned on because the default baud rate is 9600 baud. To fix this up, enter PR#1 from Applesoft direct mode and then enter the command:

```
[control-I] 8B
```

As soon as you enter [control-I] you will see a question mark prompt begin to flash; after you enter the "8B" command to set the baud rate to 1200 it will disappear. You can now send data to the printer in the normal way. Your problems are not over, however, because all your output will be double spaced. This happens because the printer automatically advances the paper one line after it receives a carriage return code but it advances it one more line in response to the line feed code that the //c automatically inserts after every carriage return. To disable the line feed insertion, enter the command:

```
[control-I] K
```

and everything will work fine.

You could also have entered the printer commands by using the Applesoft PRINT command within a program. Here is a line that you could execute to do this:

```
100 PRINT CHR$(4);"PR#1": PRINT CHR$(9);"8B"
    ;CHR$(9);"K";
```

Characteristics of a Communications Port

Serial port 2 is automatically configured as a communications port when the //c is first turned on. The initial characteristics of that port are as follows:

- A baud rate of 300
- A data format of 8 data bits, no parity, 1 stop bit
- A line feed character is not automatically inserted after each carriage return character.
- Output is not displayed on the video screen.

Once a communications device has been attached to port 2, you can tell the //c to get character input from it (or from the keyboard) by entering the IN#2 command. Similarly, you can send character output to it by entering the PR#2 command. After you enter an IN#2/PR#2 sequence like this, the //c is said to be in remote-control mode; in this mode the remote device has complete control over the //c and its operator can load and run programs, catalog the disk, and so on, in the very same way that the operator of the //c can.

You've got to be a little careful if both the IN#2 and PR#2 commands are active at the same time, however. In this situation, all input from the remote device will be echoed back to it. This is fine unless the external device is also echoing its input; if this is the case, the //c and the remote device will begin to play volleyball with the first character transmitted from either end. Use the [control-RESET] panic button to recover control if this happens.

The commands supported by the communications port include all those defined by the printer port and two more that are needed to use a special terminal mode that we'll discuss below. These additional commands are summarized in Table 11-2.

The communications port commands can only be entered after a PR#2 or IN#2 command has been entered and each must be preceded by a special command prefix character, [control-A]. Except for the "nnn" command, it is not necessary to press [return] after entering a command from the keyboard.

You can change the communications port prefix character using the same method used to change the printer port prefix character. For example, to change the prefix from [control-A] to [control-E], you would enter the command:

```
[control-A] [control-E]
```

You should note, however, that the prefix should not be changed to [control-B], [control-C], [control-H], [control-I], [control-J], [control-L], [control-M], or [control-Y]. These control characters are reserved for use by the //c's firmware.

Terminal Mode

We've just seen that the communications port supports two extra commands that relate to something called terminal mode. Terminal mode is just a fancy name for a short program in the //c's firmware that permits the //c to have a dialog with the remote device without interfering with Applesoft or

Table 11-2. Additional commands supported by the communications port.

Command	Description
Q	Exit terminal mode.
T	Enter terminal mode. This command must be entered after an IN#2 command only. If you want all incoming data to be echoed to the sender, then you must follow the IN#2 command with a PR#2 command. Here are the two full command sequences:
IN#2 [return] [control-A] T	(doesn't echo input)
IN#2 [return] PR#2 [return] [control-A] T	(echoes input)

ProDOS. While you are in terminal mode, the characters that you type in or that are sent to you by the remote device are displayed on the screen but are not passed through to the underlying operating system.

To get into terminal mode, first activate remote control mode by entering IN#2 (followed by PR#2 if you want to echo all input to output), and then enter [control-A] T. When terminal mode is active, a blinking underline cursor will appear and you can begin to converse with the remote device by typing in messages from the keyboard. Since these messages are not passed through to Applesoft or ProDOS, you can type in anything you want and you will not see an Applesoft ?SYNTAX ERROR message or overwrite the program in memory.

To exit terminal mode, you can either enter [control-A] Q from the //c's keyboard or wait for a [control-R] to be sent by the remote device. The remote device can also pop the //c back into terminal mode from remote-control mode by sending a [control-T] character.

Changing the Default Configuration

The default configurations for the two serial ports are defined by two sets of four bytes in the //c's ROM. When the //c is first turned on, these bytes are transferred to screen holes in auxiliary memory; they are read from here whenever a serial port is activated with a PR# or IN# command. The locations used to store the configuration bytes for each port and a functional description of each byte can be found in Table 11-3.

The first two bytes in each quartet contain the values that must be stored in the 6551's control and command registers in order to select the desired baud rate and data format.

Three bits in the third byte are used as flags to set whether the port is to send output to the video display as well as the external serial device, whether it is to automatically insert line feeds after carriage returns, and whether the port is a communications port or a printer port.

The last byte holds the printer width byte and will normally be zero for a communications port.

You can easily redefine a port's default configuration by storing the appropriate values in the auxiliary memory screen holes. For example, if you want port 1 to be configured as a communications port when it is initialized, just store a byte at \$47A that has bit 0 equal to 1. This configuration change will persist until the //c's power is turned off, even if another diskette is booted.

When changing the configuration bytes you must keep in mind that it is bytes in auxiliary memory, not main memory, that must be accessed. To change these bytes you will have to write a small program that first throws the 80STOREON (\$C001) switch, turns on auxiliary memory by writing to

Table 11-3. Configuration bytes used by the serial ports.

Location		Description										
Hex	(Dec)											
\$478	(1144)	Port 1: Contents of 6551 control register (Default = \$9E : 8 data bits, 2 stop bits, 9600 baud)										
\$479	(1145)	Port 1: Contents of 6551 command register (Default = \$0B : no parity)										
\$47A	(1146)	Port 1: Flags (Default = \$40 : no echo, LF after CR, printer port)										
		<table><tr><th>Bit</th><th>Bit = 1 means . . .</th></tr><tr><td>7</td><td>Echo output on video screen.</td></tr><tr><td>6</td><td>Insert line feed after carriage return.</td></tr><tr><td>5-1</td><td>[not used]</td></tr><tr><td>0</td><td>Configure as a communications port (0 = configure as a printer port).</td></tr></table>	Bit	Bit = 1 means . . .	7	Echo output on video screen.	6	Insert line feed after carriage return.	5-1	[not used]	0	Configure as a communications port (0 = configure as a printer port).
Bit	Bit = 1 means . . .											
7	Echo output on video screen.											
6	Insert line feed after carriage return.											
5-1	[not used]											
0	Configure as a communications port (0 = configure as a printer port).											
\$47B	(1147)	Port 1: Number of non-carriage-return characters to send before automatically sending a carriage return ("printer width"). If zero, then don't insert carriage returns. (Default = \$50 : 80 columns)										
\$47C	(1148)	Port 2: Contents of 6551 control register (Default = \$16 : 8 data bits, 1 stop bits, 300 baud)										
\$47D	(1149)	Port 2: Contents of 6551 command register (Default = \$0B : no parity)										
\$47E	(1150)	Port 2: Flags (Default = \$01 : echo on, no LF after CR, communications port)										
		<table><tr><th>Bit</th><th>Bit = 1 means . . .</th></tr><tr><td>7</td><td>Echo output on video screen.</td></tr><tr><td>6</td><td>Insert line feed after carriage return.</td></tr><tr><td>5-1</td><td>[not used]</td></tr><tr><td>0</td><td>Configure as a communications port (0 = configure as a printer port).</td></tr></table>	Bit	Bit = 1 means . . .	7	Echo output on video screen.	6	Insert line feed after carriage return.	5-1	[not used]	0	Configure as a communications port (0 = configure as a printer port).
Bit	Bit = 1 means . . .											
7	Echo output on video screen.											
6	Insert line feed after carriage return.											
5-1	[not used]											
0	Configure as a communications port (0 = configure as a printer port).											
\$47F	(1151)	Port 2: Number of non-carriage-return characters to send before automatically sending a carriage return ("printer width"). If zero, then don't insert carriage returns. (Default = \$00 : no CR insertion)										

Note: All locations are in the auxiliary memory screen holes.

PAGE2ON (\$C055), POKEs the new configuration bytes into the screen holes, and then re-enables main memory by writing to PAGE2OFF (\$C054). An example of such a program is given in Table 11-4. This program configures port 1 as a 1200 baud communications port and sets up a data format of 8 data bits, no parity, and one stop bit.

Table 11-4. CHANGE.PORT—a program to change the default configurations of the //c's serial ports.

```

0  REM "CHANGE.PORT"
1  REM THIS PROGRAM CHANGES THE
2  REM STARTUP CONFIGURATION OF
3  REM SERIAL PORT 1 OR 2
100 PN = 1: REM SERIAL PORT 1
110 AD = 0: IF PN = 2 THEN AD =
    4
120 POKE 49153,0: REM 80STOREON

130 POKE 49237,0: REM SELECT AU
    XILIARY MEMORY
140 POKE 1144 + AD,24: REM 8N1,
    1200 BAUD
150 POKE 1145 + AD,11: REM NO P
    ARITY
160 POKE 1146 + AD,1: REM COMM.
    PORT
170 POKE 1147 + AD,0: REM NO CR
    INSERTION
180 POKE 49236,0: REM SELECT MA
    IN MEMORY

```

6551 Interrupt Handling

The 6551 ACIA can also be programmed to generate 65C02 IRQ interrupts in the following situations:

- When the receiver register is full
- When the transmitter register is empty
- When the DCD line changes state
- When the DSR line changes state

These interrupts will be recognized and acted upon by the 65C02 only if the interrupt flag in the processor status register is 0. This condition can be forced by executing a CLI (clear interrupt) instruction.

To permit the 6551 to generate interrupts, you must always ensure that its DTR line is in a low state. This can be done by storing a 1 in bit 0 of the 6551 command register. In addition, you must store a 0 in bit 1 of the command register to enable receiver interrupts, or a 0 in bit 3 and a 1 in bit 2 of the command register to enable transmitter interrupts. (The DCD and DSR interrupts cannot be selectively disabled and enabled.) When an interrupt occurs, bit 7 of the status register will be set to 1.

We saw in Chapter 10 that the //c's ROM contains a complex interrupt-handling subroutine that is responsible for managing mouse and VBL interrupts. It's no surprise, then, to learn that it also contains an equally complex subroutine to handle interrupts emanating from the serial ports.

The //c will either handle the serial interrupt internally or will pass it along to your own interrupt-handling subroutine. It decides what to do with an interrupt by examining the types of serial interrupts that have been enabled and the contents of special flag bytes stored in the screen holes of the //c's main memory space. A flowchart of the //c's internal interrupt-handling subroutine is shown in Figure 11-6.

6551 Transmitter Interrupts

Transmitter interrupts are enabled by storing a 0 in bit 3 and a 1 in bit 2 of the 6551 command register. After this has been done, the 6551 will generate an interrupt whenever the transmitter data register becomes empty.

Transmitter interrupts are never serviced by the //c's internal interrupt handler and are always passed through to the one that you have installed. In fact, if transmitter interrupts are enabled, all 6551 interrupts are passed through.

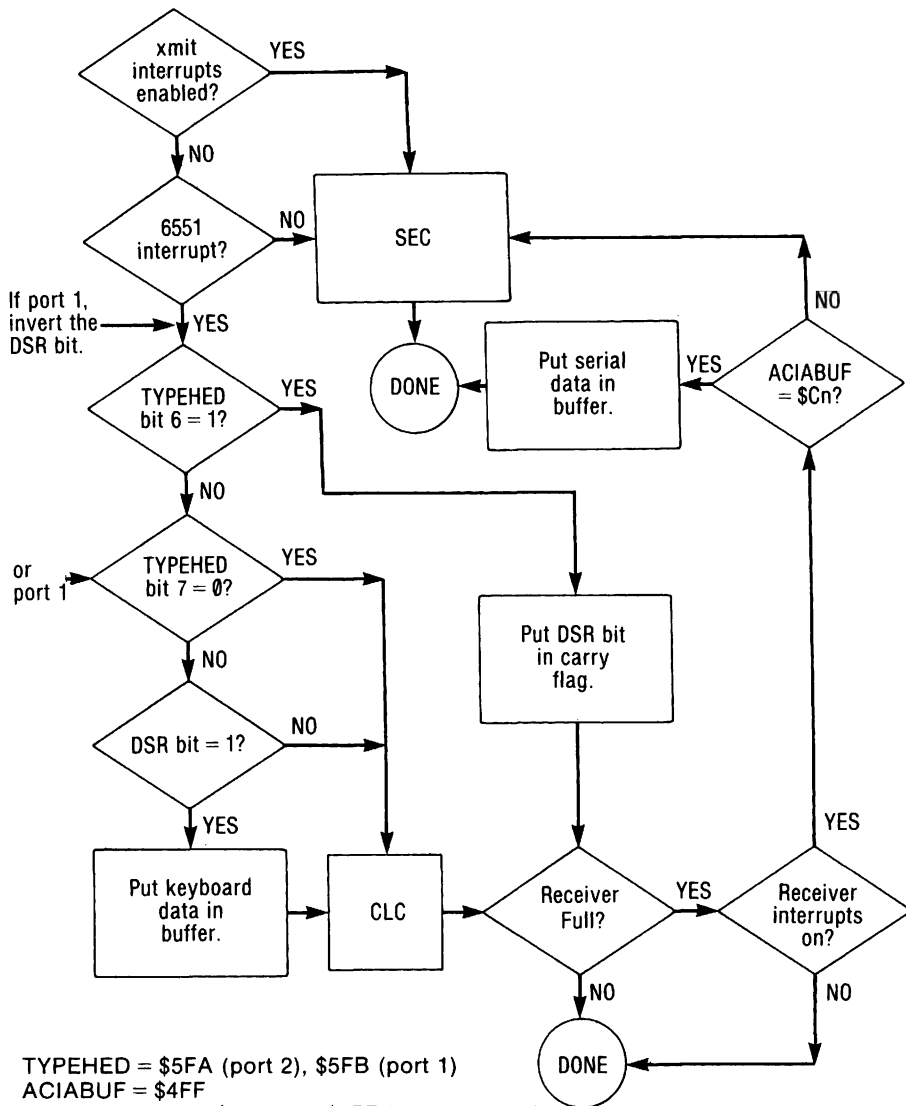
You can clear a transmitter interrupt condition by reading the 6551 status register.

6551 Receiver Interrupts

Receiver interrupts are enabled by storing a 0 in bit 1 of the 6551 control register. When this is done, the 6551 will cause an interrupt whenever the receiver data register becomes full.

The //c's internal serial interrupt handler will always pass through receiver interrupts to your own interrupt handler if transmitter interrupts have also been enabled. If transmitter interrupts have not been enabled, however, receiver interrupts will only be passed through if the value stored at ACIABUF (\$4FF) is not equal to \$Cn, where "n" is the number of the serial port that caused the interrupt (1 or 2).

If ACIABUF does contain \$Cn, then the //c services the interrupt by reading the 6551 data register and placing the data in a 128-byte receiver buffer that



TYPEHED = \$5FA (port 2), \$5FB (port 1)

ACIABUF = \$4FF

Keyboard Buffer: \$880 . . . \$8FF (aux. memory)

Serial Buffer: \$800 . . . \$87F (aux. memory)

To read serial buffer, call XRDSEB (\$C835) with Y = 0. If data is present, the carry flag will be set and the data will be in A.

On exit: If carry flag is set, interrupt passes through.

If carry flag is clear, interrupt is serviced internally.

Figure 11-6. A flowchart of the internal interrupt handler that services 6551 serial interrupts.

is located from \$800 . . . \$87F in auxiliary memory. This buffer can be read by calling the XRDSEB (\$C8C5) subroutine with the Y register set to 0 or by reading from the serial port firmware (assuming that the appropriate IN#

command is active). On exit from this subroutine, the carry flag will be clear if the buffer is empty; if it isn't, then the carry flag will be set and the character will be in the accumulator.

If a receiver interrupt is passed through, the interrupt must be serviced by reading the 6551 status register. Remember that a receiver interrupt will be passed through if it occurs when ACIABUF (\$4FF) contains a value other than \$C1 or \$C2.

The receiver interrupt is probably the most useful type of interrupt that the 6551 supports, at least as far as the //c is concerned. Why? Because at baud rates of 1200 or higher, the //c is often not capable of polling for serial input fast enough to prevent the loss of incoming characters. For example, in the time it takes the //c to scroll its full 80-column screen, two or three characters may arrive at the serial port. Since the //c's scrolling subroutine does not poll the serial ports while it executes, these characters will be missed. If receiver interrupts are enabled, however, these characters can be placed in a buffer from which they can be read when the program has the time to handle serial input.

6551 Keyboard (DSR port 2) Interrupts

The //c makes rather ingenious use of the DSR input lines on its two 6551s. The port 2 DSR line is connected to the //c's keyboard strobe line. As we saw in Chapter 7, the keyboard strobe is normally low (0) but goes high (1) when a key is pressed on the keyboard. Such a transition will cause a 6551 DSR interrupt signal to be generated from serial port 2.

The //c's built-in interrupt-handling subroutines can be told to service the keyboard interrupt and place the keycode into a 128-byte buffer in auxiliary memory (the buffer extends from \$880 to \$8FF). The //c's standard keyboard input subroutine will, in these circumstances, examine the buffer for presence of input; the keyboard I/O location, KBD (\$C000), will only be polled if the buffer is empty.

Contrast this method of handling keyboard input with the one that is traditionally used; the traditional method is to repeatedly scan the keyboard strobe line until it goes high and then read KBD (\$C000) to get the keycode. This method is called "polling" because the software is continually "asking" the keyboard whether it has a character available. One consequence of using the polling method is that if you try to enter characters from the keyboard when the //c is not actually polling the keyboard, then all those characters will be "missed" except for the last one entered.

The advantage of using the keyboard interrupt technique should be obvious: unless the interrupts are turned off by the software (using a SEI instruction), all characters entered from the keyboard will be saved in the buffer (assuming that it is not full) and will be available to the program even if the program is not, at the time of the keypress, reading the keyboard. Thus, you can "type

ahead" of the program and wait for it to read your already entered input later, when it is ready to receive it.

If you want the //c's internal serial interrupt-handling subroutines to support an interrupt-driven keyboard, then five simple steps must be performed:

- Disable 65C02 interrupts by executing a SEI instruction.
- Enable the interrupt handler's buffering of keyboard data by setting bit 7 of TYPED (\$5FA) to 1 and bit 6 to 0. (This can be done by storing \$80 in \$5FA.)
- Clear the keyboard buffer by setting locations TWKEY (\$5FF) and TRKEY (\$6FF) to \$80. TWKEY points to where the next key will be stored and TRKEY points to where the next key will be read from.
- Set DTR low by storing a 1 in bit 0 of the 6551 command register at \$C0AA (port 2).
- Enable 65C02 interrupts by executing a CLI instruction.

The short assembly-language subroutine that does all this looks something like this:

```
SEI
LDA #$80
STA $5FA
STA $5FF
STA $6FF
LDA #$01
STA $C0AA
LDA #$80
STA $5FA
CLI
RTS
```

Let's see if it works. Enter and run the following trivial program after executing the preceding subroutine:

```
100 FOR I = 1 TO 2000: NEXT
```

and then start typing madly away at the keyboard. When the program finishes, all the keystrokes that you entered should be displayed after the Applesoft prompt symbol!

By the way, the type-ahead buffer can be cleared (or "flushed") at any time by entering [control-X] from the keyboard while holding down the OPEN-APPLE key. You may want to flush the buffer in situations where you have typed in incorrect characters but the program has not yet used them.

There is one severe limitation to the type-ahead feature as implemented on the //c: it will miss characters that are typed in when the disk drive is being used. This is because ProDOS turns off interrupts during all disk accesses in order to ensure that time-critical disk I/O subroutines are not disturbed.

Keyboard interrupts can either be serviced by the //c's interrupt-handler or your own. The keyboard interrupt will be passed through to you if transmitter interrupts are enabled for port 2's 6551, or if transmitter interrupts are disabled and bit 6 of TYPHED (\$5FA) is 1.

If transmitter interrupts are enabled when a keyboard interrupt occurs, your interrupt-handling subroutine must service the interrupt by performing the following steps:

- Read the 6551 status register (\$C0A9) to check that the IRQ bit (bit 7) is "1" (that is, that the serial port is the source of the interrupt) and that the DSR bit (bit 6) is "1" (that is, that the keyboard caused the serial interrupt).
- Read KBD (\$C000) to get the keyboard data and accessing KBDSTRB (\$C010) to clear the keyboard strobe.
- Read the 6551 status register once again to clear the interrupt caused by the 1 to 0 transition on the DSR line that occurs when the keyboard strobe is cleared.

If, however, the keyboard interrupt is passed through because bit 6 of TYPHED (\$5FA) is 1 and transmitter interrupts are not enabled, the interrupt-handling subroutine cannot read the 6551 status register to check the state of the IRQ bit. This is because the //c's internal interrupt handler clears this bit by reading the status register itself before relinquishing control. Fortunately, however, it stores a copy of the value read from the status register at location \$4FA so that it can be examined instead. Before the interrupt-handling subroutine ends it must clear the interrupt by storing a 0 in \$4FA.

6551 External (DSR port 1) Interrupts

The port 1 DSR line is connected to another line that comes in through pin 9 on the //c's external disk drive connector. It is not currently used by the standard external disk drive supplied by Apple, and its intended use is still a mystery. To enable the interrupt signal that is generated when this line changes state, bit 0 of the 6551 control register for port 1 (\$C09A) must be set to 1.

The //c's serial interrupt-handler monitors the state of the external DSR interrupt line but does not do much with it. If transmitter interrupts are enabled for port 1's 6551, however, or if they're not but bit 6 of EXTINT2 (\$5F9) is 1, the interrupt will be passed on through to your own interrupt-handling subroutine so that you can deal with it yourself.

Note that the //c has been configured in such a way that even if bit 6 of EXTINT2 is 1, a port 1 DSR interrupt will only be passed through when high (1) to low (0) transitions of the DSR line take place (just the opposite to port 2). From this we can assume that any device that generates an external

interrupt will be expected to keep the interrupt line high and to bring it low only when an interrupt occurs.

The port 2 DSR interrupt can be serviced in the same general way as the corresponding interrupt for port 1. First, you must read the status register to check the state of DSR and clear the IRQ bit. Then, you must deal with the device that caused the interrupt in such a way that its interrupt signal will be turned off. (Without knowing what the device is we can't really say what to do.) Finally, the status register must be read once again in order to clear the interrupt caused by the reverse transition of the external interrupt line after the device turns off its interrupt signal.

Further Reading for Chapter 11

On serial communications in general . . .

E.A. Nichols, J.C. Nichols, and K.R. Musson, *Data Communications for Microcomputers*, 1982, McGraw-Hill Book Company. This book describes the RS-232-C standard in detail as well as various transmission protocols.

Appendix I

American National Standard Code for Information Interchange (ASCII) Character Codes

<i>Hex</i>	<i>ASCII Code</i>	<i>Dec</i>	<i>Symbol</i>	<i>Keys to Press</i>
\$00		000	NUL	(Null)
\$01		001	SOH	(Start of header)
\$02		002	STX	(Start of text)
\$03		003	ETX	(End of text)
\$04		004	EOT	(End of transmission)
\$05		005	ENQ	(Enquiry)
\$06		006	ACK	(Acknowledge)
\$07		007	BEL	(Bell)
\$08		008	BS	(Backspace)
\$09		009	HT	(Horizontal tabulation)
\$0A		010	LF	(Line feed)
\$0B		011	VT	(Vertical tabulation)
\$0C		012	FF	(Form feed)
\$0D		013	CR	(Carriage return)
\$0E		014	SO	(Shift out)
\$0F		015	SI	(Shift in)
\$10		016	DLE	(Data link escape)
\$11		017	DC1	(Device control 1)
\$12		018	DC2	(Device control 2)
\$13		019	DC3	(Device control 3)
\$14		020	DC4	(Device control 4)
\$15		021	NAK	(Negative acknowledge)
\$16		022	SYN	(Synchronous idle)
\$17		023	ETB	(End of transmission block)
\$18		024	CAN	(Cancel)
\$19		025	EM	(End of medium)
\$1A		026	SUB	(Substitute)
\$1B		027	ESC	(Escape)
\$1C		028	FS	(Field separator)

\$1D	GS	(Group separator)	CONTROL]
\$1E	RS	(Record separator)	CONTROL ^
\$1F	US	(Unit separator)	CONTROL _
\$20	!	(Space)	SPACE BAR
\$21	"		SHIFT 1
\$22	#		SHIFT ,
\$23	\$		SHIFT 3
\$24	%		SHIFT 4
\$25	&		SHIFT 5
\$26	,		SHIFT 7
\$27	(,
\$28)		SHIFT 9
\$29	*		SHIFT 0
\$2A	+		SHIFT 8
\$2B	.		SHIFT =
\$2C	/		.
\$2D	0		/
\$2E	1		0
\$2F	2		1
\$30	3		2
\$31	4		3
\$32	5		4
\$33	6		5
\$34	7		6
\$35	8		7
\$36	9		8
\$37	:		9
\$38	;		SHIFT ;
\$39	.		,
\$3A	<		SHIFT ,
\$3B			
\$3C			(continued)

ASCII Code		Symbol	Keys to Press
Hex	Dec		
\$3D	061	=	SHIFT .
\$3E	062	>	SHIFT /
\$3F	063	?	SHIFT 2
\$40	064	@	SHIFT A
\$41	065	A	SHIFT B
\$42	066	B	SHIFT C
\$43	067	C	SHIFT D
\$44	068	D	SHIFT E
\$45	069	E	SHIFT F
\$46	070	F	SHIFT G
\$47	071	G	SHIFT H
\$48	072	H	SHIFT I
\$49	073	I	SHIFT J
\$4A	074	J	SHIFT K
\$4B	075	K	SHIFT L
\$4C	076	L	SHIFT M
\$4D	077	M	SHIFT N
\$4E	078	N	SHIFT O
\$4F	079	O	SHIFT P
\$50	080	P	SHIFT Q
\$51	081	Q	SHIFT R
\$52	082	R	SHIFT S
\$53	083	S	SHIFT T
\$54	084	T	SHIFT U
\$55	085	U	SHIFT V
\$56	086	V	SHIFT W
\$57	087	W	SHIFT X
\$58	088	X	SHIFT Y
\$59	089	Y	



SHIFT Z
 [\] SHIFT 6
 SHIFT - , A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(continued)

Z [\] ^ _ , a b c d e f g h i j k l m n o p q r s t u v w x y z

\$090
 \$091
 \$092
 \$093
 \$094
 \$095
 \$096
 \$097
 \$098
 \$099
 \$100
 \$101
 \$102
 \$103
 \$104
 \$105
 \$106
 \$107
 \$108
 \$109
 \$110
 \$111
 \$112
 \$113
 \$114
 \$115
 \$116
 \$117
 \$118
 \$119
 \$120
 \$121
 \$122

\$5A
 \$5B
 \$5C
 \$5D
 \$5E
 \$5F
 \$60
 \$61
 \$62
 \$63
 \$64
 \$65
 \$66
 \$67
 \$68
 \$69
 \$6A
 \$6B
 \$6C
 \$6D
 \$6E
 \$6F
 \$70
 \$71
 \$72
 \$73
 \$74
 \$75
 \$76
 \$77
 \$78
 \$79
 \$7A



ASCII Code		Symbol	Keys to Press
Hex	Dec		
\$7B	123	{	SHIFT [
\$7C	124		SHIFT \
\$7D	125	}	SHIFT]
\$7E	126	~	SHIFT '
\$7F	127	■ (Rubout)	DELETE

Appendix II

65C02 Instruction Set and Cycle Times

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
ADC	#num	69	2	2	
	zpage	65	2	3	
	zpage,X	75	2	4	
	*(zpage)	72	2	5	
	(zpage,X)	61	2	6	
	(zpage),Y	71	2	5	(1)
	abs	6D	3	4	
	abs,X	7D	3	4	(1)
	abs,Y	79	3	4	(1)
AND	#num	29	2	2	
	zpage	25	2	3	
	zpage,X	35	2	4	
	*(zpage)	32	2	5	
	(zpage,X)	21	2	6	
	(zpage),Y	31	2	5	(1)
	abs	2D	3	4	
	abs,X	3D	3	4	(1)
	abs,Y	39	3	4	(1)
ASL	[accumulator]	0A	1	2	
	zpage	06	2	5	
	zpage,X	16	2	6	
	abs	0E	3	6	
	abs,X	1E	3	6	(3)
BCC	disp	90	2	2	(2)
BCS	disp	B0	2	2	(2)
BEQ	disp	F0	2	2	(2)

(continued)

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
BIT	*#num	89	2	2	
	zpage	24	2	3	
	*zpage,X	34	2	4	
	abs	2C	3	4	
	*abs,X	3C	3	4	
BMI	disp	30	2	2	(2)
BNE	disp	D0	2	2	(2)
BPL	disp	10	2	2	(2)
BRA	*disp	80	2	2	(2)
BRK	[implied]	00	1	7	
BVC	disp	50	2	2	(2)
BVS	disp	70	2	2	(2)
CLC	[implied]	18	1	2	
CLD	[implied]	D8	1	2	
CLI	[implied]	58	1	2	
CLV	[implied]	B8	1	2	
CMP	#num	C9	2	2	
	zpage	C5	2	3	
	zpage,X	D5	2	4	
	*(zpage)	D2	2	5	
	(zpage,X)	C1	2	6	
	(zpage),Y	D1	2	5	(1)
	abs	CD	3	4	
	abs,X	DD	3	4	(1)
	abs,Y	D9	3	4	(1)
CPX	#num	E0	2	2	
	zpage	E4	2	3	
	abs	EC	3	4	
CPY	#num	C0	2	2	
	zpage	C4	2	3	
	abs	CC	3	4	
DEA	*[accumulator]	3A	1	2	
DEC	zpage	C6	2	5	
	zpage,X	D6	2	6	
	abs	CE	3	6	
	abs,X	DE	3	6	(3)
DEX	[implied]	CA	1	2	

(continued)

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
DEY	[implied]	88	1	2	
EOR	#num	49	2	2	
	zpage	45	2	3	
	zpage,X	55	2	4	
	*(zpage)	52	2	5	
	(zpage,X)	41	2	6	
	(zpage),Y	51	2	5	(1)
	abs	4D	3	4	
	abs,X	5D	3	4	(1)
	abs,Y	59	3	4	(1)
INA	*[accumulator]	1A	1	2	
INC	zpage	E6	2	5	
	zpage,X	F6	2	6	
	abs	EE	3	6	
	abs,X	FE	3	6	(3)
INX	[implied]	E8	1	2	
INY	[implied]	C8	1	2	
JMP	abs	4C	3	3	
	(abs)	6C	3	6	(4)
	*(abs,X)	7C	3	6	
JSR	abs	20	3	6	
LDA	#num	A9	2	2	
	zpage	A5	2	3	
	zpage,X	B5	2	4	
	*(zpage)	B2	2	5	
	(zpage,X)	A1	2	6	
	(zpage),Y	B1	2	5	(1)
	abs	AD	3	4	
	abs,X	BD	3	4	(1)
	abs,Y	B9	3	4	(1)
LDX	#num	A2	2	2	
	zpage	A6	2	3	
	zpage,Y	B6	2	4	
	abs	AE	3	4	
	abs,Y	BE	3	4	(1)
LDY	#num	A0	2	2	
	zpage	A4	2	3	
	zpage,X	B4	2	4	
	abs	AC	3	4	
	abs,X	BC	3	4	(1)

(continued)

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
LSR	[accumulator]	4A	1	2	(3)
	zpage	46	2	5	
	zpage,X	56	2	6	
	abs	4E	3	6	
	abs,X	5E	3	6	
NOP	[implied]	EA	1	2	
ORA	#num	09	2	2	(1)
	zpage	05	2	3	
	zpage,X	15	2	4	
	*(zpage)	12	2	5	
	(zpage,X)	01	2	6	
	(zpage),Y	11	2	5	
	abs	0D	3	4	
	abs,X	1D	3	4	
	abs,Y	19	3	4	
PHA	[implied]	48	1	3	
PHP	[implied]	08	1	3	
PHX	*[implied]	DA	1	3	
PHY	*[implied]	5A	1	3	
PLA	[implied]	68	1	4	
PLP	[implied]	28	1	4	
PLX	*[implied]	FA	1	4	
PLY	*[implied]	7A	1	4	
ROL	[accumulator]	2A	1	2	(3)
	zpage	26	2	5	
	zpage,X	36	2	6	
	abs	2E	3	6	
	abs,X	3E	3	6	
ROR	[accumulator]	6A	1	2	(3)
	zpage	66	2	5	
	zpage,X	76	2	6	
	abs	6E	3	6	
	abs,X	7E	3	6	
RTI	[implied]	40	1	6	
RTS	[implied]	60	1	6	
SBC	#num	E9	2	2	
	zpage	E5	2	3	
	zpage,X	F5	2	4	

(continued)

<i>Instruction Mnemonic</i>	<i>Assembler Operand Format</i>	<i>Opcode Byte</i>	<i>Number of Bytes</i>	<i>Number of Clock Cycles</i>	<i>Notes</i>
	*(zpage)	F2	2	5	
	(zpage,X)	E1	2	6	
	(zpage),Y	F1	2	5	(1)
	abs	ED	3	4	
	abs,X	FD	3	4	(1)
	abs,Y	F9	3	4	(1)
SEC	[implied]	38	1	2	
SED	[implied]	F8	1	2	
SEI	[implied]	78	1	2	
STA	zpage	85	2	3	
	zpage,X	95	2	4	
	*(zpage)	92	2	5	
	(zpage,X)	81	2	6	
	(zpage),Y	91	2	5	(1)
	abs	8D	3	4	
	abs,X	9D	3	4	(1)
	abs,Y	99	3	4	(1)
STX	zpage	86	2	3	
	zpage,Y	96	2	4	
	abs	8E	3	4	
STY	zpage	84	2	3	
	zpage,X	94	2	4	
	abs	8C	3	4	
STZ	*zpage	64	2	3	
	*zpage,X	74	2	4	
	*abs	9C	3	4	
	*abs,X	9E	3	5	
TAX	[implied]	AA	1	2	
TAY	[implied]	A8	1	2	
TRB	*zpage	14	2	5	
	*abs	1C	3	6	
TSB	*zpage	04	2	5	
	*abs	0C	3	6	
TSX	[implied]	BA	1	2	
TXA	[implied]	8A	1	2	
TXS	[implied]	9A	1	2	
TYA	[implied]	98	1	2	

(continued)

*Instructions marked with an asterisk are not available on the 6502.

Notes:

- (1) Add one clock cycle if a page boundary is crossed.
- (2) Add one clock cycle if a branch occurs to a location in the same page; add two clock cycles if a branch occurs to a location in a different page.
- (3) Add one clock cycle if a page boundary is crossed; always 7 cycles on the 6502.
- (4) 5 cycles on the 6502.

See Table 2-3 for a description of the assembler operand formats.

Appendix III

Apple //c Soft Switch, Status, and I/O Port Locations

NOTE: The “Usage” column in the following tables indicates how a particular location is to be accessed:

“W” means “write to the location.”

“R” means “read from the location.”

“RW” means “read from or write to the location.”

“R7” means “read and check bit 7 to determine the status.”

“RR” means “read from the location twice in succession.”

The term “aux.” refers to the auxiliary block of 64K memory and “main” refers to the main block of 64K memory. “BSR” refers to the //c’s 16K bank-switched RAM space from \$D000-\$FFFF.

Address Hex	(Dec)	Usage	Symbolic Name	Action Taken	Note
\$C000	(49152)	R	KBD	Keyboard data (bits 0 . . . 6)	
		R7	KBD	1 = keyboard strobe is on 0 = keyboard strobe is off	
\$C000	(49152)	W	80STOREOFF	Allow PAGE2 to switch between video page1 and page2	1
\$C001	(49153)	W	80STOREON	Allow PAGE2 to switch between main and aux. video memory	1
\$C002	(49154)	W	RAMRDON	Read-enable main memory from \$200-\$BFFF	4
\$C003	(49155)	W	RAMRDON	Read-enable aux. memory from \$200-\$BFFF	4
\$C004	(49156)	W	RAMWRTOFF	Write-enable main memory from \$200-\$BFFF	4
\$C005	(49157)	W	RAMWRTON	Write-enable aux. memory from \$200-\$BFFF	4
\$C006	(49158)			[Reserved]	
\$C007	(49159)			[Reserved]	
\$C008	(49160)	W	ALTZPOFF	Enable main memory from \$0000-\$01FF and make main BSR available	
\$C009	(49161)	W	ALTZPON	Enable aux. memory from \$0000-\$01FF and make aux. BSR available	
\$C00A	(49162)			[Reserved]	
\$C00B	(49163)			[Reserved]	
\$C00C	(49164)	W	80COLOFF	Turn off 80-column display	
\$C00D	(49165)	W	80COLON	Turn on 80-column display	
\$C00E	(49166)	W	ALTCHARSETOFF	Turn off alternative characters	
\$C00F	(49167)	W	ALTCHARSETON	Turn on alternative characters	
\$C010	(49168)	RW	KBDSTRB	Clear keyboard strobe	
		R7	AKD	1 = a key is being pressed 0 = all keys are released	3
\$C011	(49169)	R7	RDBANK2	1 = bank2 of BSR is available 0 = bank1 of BSR is available	
\$C012	(49170)	R7	RDLCRAM	1 = BSR is active for read operations 0 = \$D000-\$FFFF ROM is active for read operations	

\$C013	(49171)	R7	RAMRD	1 = aux. \$200-\$BFFF is active for read operations 0 = main \$200-\$BFFF is active for read operations	4
\$C014	(49172)	R7	RAMWRT	1 = aux. \$200-\$BFFF is active for write operations 0 = main \$200-\$BFFF is active for write operations	4
\$C015	(49173)	R7	MOUSEXINT	1 = mouse X0 interrupt has occurred 0 = no mouse X0 interrupt	
\$C016	(49174)	R7	ALTZP	1 = aux. zero page + stack is active; aux. BSR is available 0 = main zero page + stack is active; main BSR is available	
\$C017	(49175)	R7	MOUSEYINT	1 = mouse Y0 interrupt has occurred 0 = no mouse Y0 interrupt	
\$C018	(49176)	R7	80STORE	1 = PAGE2 switches main/aux. 0 = PAGE2 switches video pages	1
\$C019	(49177)	R7	VBLINT	1 = a VBL interrupt has occurred 0 = no VBL interrupt	
\$C01A	(49178)	R7	TEXT	1 = a text mode is active 0 = a graphics mode active	
\$C01B	(49179)	R7	MIXED	1 = mixed graphics and text 0 = full-screen graphics	2
\$C01C	(49180)	R7	PAGE2	1 = video page2 selected OR aux. video page selected	1
\$C01D	(49181)	R7	HIRES	1 = high-resolution graphics 0 = low-resolution graphics	1,2
\$C01E	(49182)	R7	ALTCHARSET	1 = alternative character is on 0 = primary character is on	
\$C01F	(49183)	R7	80COL	1 = 80-column display is on 0 = 40-column display is on	

(continued)

Address		Usage	Symbolic Name	Action Taken	Note
Hex	(Dec)				
\$C020	(49184)			[Reserved]	
	through				
\$C02F	(49199)				
\$C030	(49200)				
	through				
\$C03F	(49215)	R	SPEAKER	Toggle the state of the speaker	
\$C040	(49216)				
\$C041	(49217)	R7	RDXYMSK	1 = mouse interrupts enabled 0 = mouse interrupts disabled	
\$C042	(49218)	R7	RDVBLMSK	1 = VBL interrupts enabled 0 = VBL interrupts disabled	
\$C043	(49219)	R7	RDX0EDGE	1 = interrupt on falling X0 edge 0 = interrupt on rising X0 edge	
\$C044	(49220)			1 = interrupt on falling Y0 edge 0 = interrupt on rising Y0 edge	
\$C045	(49221)			[Reserved]	
\$C046	(49222)			[Reserved]	
\$C047	(49223)			[Reserved]	
\$C048	(49224)			Clear X0/Y0 mouse interrupt condition	
\$C049	(49225)	R	RSTXY	[Reserved]	
\$C04A	(49226)			[Reserved]	
\$C04B	(49227)			[Reserved]	
\$C04C	(49228)			[Reserved]	
\$C04D	(49229)			[Reserved]	
\$C04E	(49230)			[Reserved]	
\$C04F	(49231)			[Reserved]	
\$C050	(49232)	RW	TEXTOFF	Select graphics mode	
\$C051	(49233)	RW	TEXTON	Select text mode	

\$C052	(49234)	RW	MIXEDOFF	Use full screen for graphics	2
\$C053	(49235)	RW	MIXEDON	Use graphics with four lines of text	2
\$C054	(49236)	RW	PAGE2OFF	Select page1 display (or main video memory)	1
\$C055	(49237)	RW	PAGE2ON	Select page2 display (or aux. video memory)	1
\$C056	(49238)	RW	HIRESOFF	Select low-resolution graphics	1,2
\$C057	(49239)	RW	HIRESON	Select high-resolution graphics	1,2

[The action for the following soft switches from \$C058 ... \$C05F is only taken if access has first been enabled by writing to IOUDISOFF (\$C07F).]

\$C058	(49240)	RW	DISXY	Disable mouse X0/Y0 interrupts
\$C059	(49241)	RW	ENBXY	Enable mouse X0/Y0 interrupts
\$C05A	(49242)	RW	DISVBL	Disable VBL interrupts
\$C05B	(49243)	RW	ENVBL	Enable VBL interrupts
\$C05C	(49244)	RW	RX0EDGE	Interrupt on rising mouse X0
\$C05D	(49245)	RW	FX0EDGE	Interrupt on falling mouse X0
\$C05E	(49246)	RW	RY0EDGE	Interrupt on rising mouse Y0
\$C05F	(49247)	RW	FY0EDGE	Interrupt on falling mouse Y0

[The action for the following soft switches from \$C058 ... \$C05F is only taken if access has first been enabled by writing to IOUDISON (\$C07E).]

\$C058	(49240)	[Reserved]		
through				
\$C05D	(49245)			Enable double high-resolution
\$C05E	(49246)	RW	DHIRESON	
\$C05F	(49247)	RW	DHIRESOFF	Disable double high-resolution
\$C060	(49248)	R7	RD80SW	1 = 40/80 switch is down 0 = 40/80 switch is up
\$C061	(49249)	R7	PB0	1 = push button 0 or OPEN-APPLE is pressed
\$C062	(49250)	R7	PB1	1 = push button 1 or SOLID-APPLE is pressed
\$C063	(49251)	R7	RD63	1 = mouse button is not pressed 0 = mouse button is pressed
\$C064	(49252)	R7	PDL0	1 = game controller 0 not timed out (continued)

Address Hex	(Dec)	Usage	Symbolic Name	Action Taken	Note
\$C065	(49253)	R7	PDL1	1 = game controller 1 not timed out	
\$C066	(49254)	R7	MOUX1	1 = mouse has moved to right	
\$C067	(49255)	R7	MOUY1	1 = mouse has moved up	
\$C068	(49256)			[Reserved]	
\$C06F	(49263)				
\$C070	(49264)	R	PTRIG	Reset the game controllers and clear the VBL interrupt condition	
\$C071	(49265)			[Reserved]	
\$C077	(49271)	R7		[Same as \$C07E]	
\$C078	(49272)	W		[Same as \$C07E]	
\$C079	(49273)	R7		[Same as \$C07F]	
		W		[Same as \$C07F]	
\$C07A	(49274)	R7		[Same as \$C07E]	
		W		[Same as \$C07E]	
\$C07B	(49275)	R7		[Same as \$C07F]	
		W		[Same as \$C07F]	
\$C07C	(49276)	R7		[Same as \$C07E]	
		W		[Same as \$C07E]	
\$C07D	(49277)	R7		[Same as \$C07F]	
		W		[Same as \$C07F]	
\$C07E	(49278)	R7	RDIUDIS	1 = IOU access is off	
				0 = IOU access is on	
		W	IOUDISON	Disable \$C058-\$C05F IOU access and enable the DHIREs switches.	
\$C07F	(49279)	R7	DHIREs	1 = double high-res is on	

W

IOUDISOFF

Enable \$C058-\$C05F IOU access and disable the
DHIREs switches.

Notes:

1. If 80STORE is ON, then PAGE2OFF activates main video RAM (\$400-\$7FF) and PAGE2ON activates auxiliary video RAM. If HIREs is also ON, then PAGE2OFF also activates main high-resolution video RAM (\$2000-\$3FFF) and PAGE2ON also activates auxiliary high-resolution video RAM.
If 80STORE is OFF, then PAGE2OFF turns on text page1 mode and PAGE2 turns on text page2 mode. If HIREs is also ON, then PAGE2OFF also selects high-resolution page1 mode and PAGE2ON selects high-resolution page2 mode.
2. The HIREs and MIXED switches are meaningful only if the TEXT switch is OFF (that is, a graphics mode is active).
3. Reading this switch will cause the keyboard strobe (bit 7 of \$C000) to be cleared.
4. The RAMRD and RAMWRT switches do not affect the video RAM area from \$400-\$7FF if the 80STORE switch is ON or the high-resolution graphics area from \$2000-\$3FFF if the HIREs switch is ON as well. In these situations, these RAM areas are controlled by the PAGE2 switches.

I/O Port Locations

Address		Usage	Symbolic Name	Action Taken
Hex	(Dec)			
\$C080	(49280)	R	READBSR2	Select Bank2, read BSR, write-protect BSR
\$C081	(49281)	RR	WRITEBSR2	Select Bank2, read ROM, write-enable BSR
\$C082	(49282)	R	OFFBSR2	Select Bank2, read ROM, write-protect BSR
\$C083	(49283)	RR	RDWRBSR2	Select Bank2, read BSR, write-enable BSR
\$C084	(49284)			[Reserved]
through				
\$C087	(49287)			
\$C088	(49288)	R	READBSR1	Select Bank1, read BSR, write-protect BSR
\$C089	(49289)	RR	WRITEBSR1	Select Bank1, read ROM, write-enable BSR
\$C08A	(49290)	R	OFFBSR1	Select Bank1, read ROM, write-protect BSR
\$C08B	(49291)	RR	RDWRBSR1	Select Bank1, read BSR, write-enable BSR

(continued)

I/O Port Locations					Note
Hex	Address (Dec)	Usage	Symbolic Name	Action Taken	
\$C08C	(49292)				
	through				
\$C08F	(49295)			[Reserved]	
\$C090	(49296)				
	through				
\$C097	(49303)				
\$C098	(49304)				
\$C099	(49305)				
		R	DATAREG1	6551 receive data register (port 1)	
\$C09A	(49306)	W	DATAREG1	6551 transmit data register (port 1)	
\$C09B	(49307)	R	STATUS1	6551 status register (port 1)	
\$C09C	(49308)	W	RESET1	6551 programmed reset (port 1)	
		RW	COMMAND1	6551 command register (port 1)	
		RW	CONTROL1	6551 control register (port 1)	
\$C09F	(49311)			[Reserved]	
\$C0A0	(49312)				
	through				
\$C0A7	(49319)				
\$C0A8	(49320)				
\$C0A9	(49321)				
		R	DATAREG2	6551 receive data register (port 2)	
		W	DATAREG2	6551 transmit data register (port 2)	
		R	STATUS2	6551 status register (port 2)	
		W	RESET2	6551 programmed reset (port 2)	
\$C0AA	(49322)	RW	COMMAND2	6551 command register (port 2)	
\$C0AB	(49323)	RW	CONTROL2	6551 control register (port 2)	
\$C0AC	(49324)				
	through				
\$C0AF	(49327)			[Reserved]	

\$C0B0 (49328) through \$C0DF (49375)	[Reserved] [Reserved] [Reserved]
\$C0E0 (49376) through \$C0EF (49391) \$C0F0 (49392) through \$C0FF (49408)	Used for disk drive control (port 6) [Reserved]

Appendix IV

Apple //c Page 3 Vectors

<i>Address</i>	<i>Contents</i>	<i>Description</i>
\$3D0-\$3D2	JMP \$BE00	A JMP instruction to the ProDOS warm-start entry point. A call to this vector will reconnect DOS without destroying the Applesoft program in memory. Use the "3D0G" command to move from the system monitor to Applesoft.
\$3D3-\$3D5	JMP \$BE00	A JMP instruction to the ProDOS warm-start entry point.
\$3D6-\$3EC		[Reserved by ProDOS]
\$3ED-\$3EE		The address of the subroutine to be called by XFER (\$C314) is stored here.
\$3EF		[Reserved by ProDOS]
\$3F0-\$3F1	\$FA59	The address of the subroutine to which control is to be passed when a BRK instruction is executed (low-order byte first).
\$3F2-\$3F3	\$BE00	The address of the subroutine to which control is to be passed when a RESET interrupt is generated (low-order byte first).
\$3F4	\$1B	POWERED-UP BYTE. The reset vector at \$3F2 is used only if the number stored here is equal to the logical exclusive-OR of the number stored at \$3F3 and the constant \$A5.
\$3F5-\$3F7	JMP \$BE03	A JMP instruction to the subroutine to which control is to be passed when the Applesoft "&" command is executed.
\$3F8-\$3FA	JMP \$BE00	A JMP instruction to the subroutine to which control is to be passed when the system monitor's USER command ([control-Y]) is entered.

(continued)

<i>Address</i>	<i>Contents</i>	<i>Description</i>
\$3FB-\$3FD	JMP \$FF59	[Reserved by system monitor]
\$3FE-\$3FF	\$BFEB	The address of the subroutine to which control is to be passed when an IRQ interrupt is generated (low-order byte first).

Note: All addresses are stored with the low-order byte first.

Appendix V

For Beginners Only

The purpose of this appendix is to familiarize novice programmers with a few of the more important fundamental computer concepts. By mastering these concepts before reading the main body of this book, you should be able to more easily understand the technical descriptions and programming examples that will be presented.

For more detailed information on these topics, more general books on computing should be consulted. Many of the references included at the end of each chapter in this book will be useful in this regard.

Numbering Systems

We are all familiar with the decimal numbering system that makes use of ten fundamental digits. This system, however, is not sacred and we could, if we preferred, use other systems that use fewer or more digits.

When dealing with computers, it is often convenient to use the binary numbering system and the hexadecimal numbering system. The binary numbering system uses only two digits, 0 and 1. The hexadecimal system uses the following sixteen digits:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

which represent decimal numbers 0 through 15, respectively.

The 65C02 microprocessor that controls the Apple //c performs all its internal operations using binary numbers because it has available to it thousands of logic cells that can easily be turned either “on” or “off” to represent the binary digits “1” or “0,” respectively. Binary numbers, however, are usually not used when writing a program because they are difficult to read and are prone to transcription errors. Decimal-number equivalents of binary numbers are often used instead, but the pattern of binary ones and zeros to which they refer are often not immediately obvious (quick now, what is the binary representation of 225?). The hexadecimal numbering system, however, is an ideal alternative because each hexadecimal digit defines exactly one of the sixteen four-digit patterns of binary ones and zeros, making conversion between binary and hexadecimal very easy.

In this book, hexadecimal numbers will be preceded by "\$" to distinguish them from decimal numbers. They will be used when referring to data values or to memory addresses.

Bit Numbering and "Significance"

The basic unit of storage in the Apple //c, and most other microcomputers, is the byte. As far as the 65C02 microprocessor is concerned, each byte is made up of eight bits, each of which can be either on or off (a computer likes things that can exist in only one of two states). This means that binary numbers from 00000000 to 11111111 (0 to 255 decimal) can be stored in one byte.

Each bit in a byte is associated with a certain binary weight equal to the number that the byte would represent if that bit were on and all the other bits were off. These binary weights are as shown in Figure V-1.

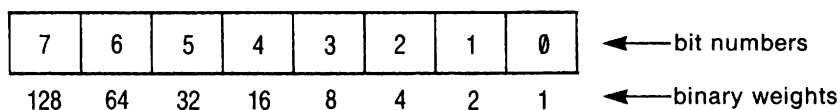


Figure V-1. Binary weights of each bit in a byte.

(Notice that the bits within the byte are numbered from 0 to 7 and not from 1 to 8.) To determine the decimal representation of the bit pattern, it is simply necessary to add up the binary weights of all bits in the byte that are on. Since bit 7 contributes most, it is called the most-significant bit or "high-order" bit. Conversely, bit 0 is referred to as the least-significant bit or "low-order" bit.

Bit 7 of a byte is also called the "sign bit" because it is often used to indicate whether the number stored in the byte is positive or negative (if it is 1, then the number is considered to be negative). The 65C02 microprocessor that controls the //c uses a special internal status register which, among other things, holds a flag that represents the sign of any number being dealt with. (See Chapter 2.) Special 65C02 instructions are available that can change the flow of a program depending on the state of this sign flag (they are called "BPL," branch on plus, and "BMI," branch on minus). The //c uses bit 7 of several special memory locations to hold information relating to the state of the system. When these status locations are examined in an assembly-language program, BPL can be used to transfer control if the status is off (bit 7 is 0) and BMI can be used to transfer control if the status is on (bit 7 is 1). The same thing can be done from an Applesoft program by using the PEEK command to read the number stored at the status location. If bit 7 is on, then the value read will be greater than or equal to 128 (since the binary weight of bit 7 is 128).

Situations where more than one byte is required to store a number (that is, the number is larger than 255) are quite common. In these cases, the byte that contains information on the highest-weighted bits for the number is called the most-significant byte or high-order byte, and the byte that contains information on the lowest-weighted bits is called the least-significant byte or low-order byte.

Pointers and Vectors

In Chapter 2 you will see that the 65C02 microprocessor is capable of controlling a memory space that is mapped to the addresses from \$0000 . . . \$FFFF. Since one byte can hold exactly two hexadecimal digits, any address in the 65C02's memory space can be stored in two bytes.

A pointer or "vector" is a pair of memory locations that contains the address of another location to which the pointer is said to be pointing. The least-significant byte of the pair is always stored in the first memory location and the other byte in the next higher location. To determine the address stored in a pointer, you can use the following Applesoft formula:

$$\text{ADDR} = \text{PEEK}(X) + 256 * \text{PEEK}(X + 1)$$

where X represents the first memory location which the pointer occupies. The second byte in the pair is multiplied by 256 since it represents the number of 256-byte units that make up the address.

The 65C02 microprocessor makes extensive use of pointers to access data arrays and to handle interrupts. (See Chapter 2.) Applesoft also maintains a great many pointers for keeping track of its many data areas. (See Chapter 4.)

Control Characters

Control characters are special characters that are entered from the keyboard by using the CONTROL key. Although they do not represent visible symbols, they often cause the //c to perform special functions. Such characters will be denoted in this book by [control-X], where X refers to any alphabetic character (A . . . Z) or one of the following special symbols: @ [\] ^ _ . The CONTROL key acts just like another SHIFT key in that it and one other key must be pressed at the same time in order to enter a control character from the keyboard. The procedure involves first pressing the CONTROL key and then, while still holding it down, pressing the other key ("X" in the above example).

65C02 Assembly Language

Many of the programs presented in this book are written in a programming language that can be used to generate a series of bytes (which represent

microprocessor instructions and data) that can be interpreted and directly executed by the //c's 65C02 microprocessor. This programming language is called "65C02 assembly language."

There are two steps involved in developing an assembly-language program. First, a source code for the program must be entered that defines the program in a human-readable form using symbolic labels for addresses and data, special three-character mnemonics for the permitted 65C02 instructions, and special symbols to indicate the addressing modes used by the instructions. (See Chapter 2 for a detailed discussion of 65C02 instructions and addressing modes.)

A typical line of source code looks something like this:

```
LABEL LDA ($28),Y ;This is a comment
```

and is made up of four distinct fields. The first field is the label field and it holds the symbolic name (if any) for the current location within the program. The next field is the instruction field and it holds the three-character mnemonic for the 65C02 instruction ("LDA" in the example). It is immediately followed by the operand field, which holds the addressing mode used by the instruction, that is, information relating to the method the instruction is to use to access the data or memory location on which it is to act, the actual address or data itself, or an expression that evaluates to that address or data ("(\$28),Y" in the example). The last field is the comment field and is used for documenting the program. Each field is separated from the other by at least one blank space; in addition, most assemblers require comments to be preceded by a semicolon.

The second step is to interpret or "assemble" the program source code using a 65C02 assembler. This is done in order to produce a file that contains the bytes defined by the program in a format that the 65C02 can directly execute (the "object code" or "machine language").

The assembly-language programs presented in this book were all entered and assembled using the Merlin Pro assembler published by Roger Wagner Publishing, Inc. (10761 Woodside Avenue, Suite E, Santee, California 92071). If you want to modify and reassemble the programs presented in this book and you are not using Merlin Pro, then you will likely have to make several changes to the program source codes to account for any differences in syntax and command structure. Differences usually arise in the area of "pseudo-instructions"; these are assembler-specific commands that appear in the 65C02 instruction field of a line of source code, but that represent commands to the assembler rather than 65C02 instructions. They can be used to place data bytes at specific locations within the program (DFB, DS, and ASC), to define symbolic labels (EQU), to indicate the starting address of the program (ORG), and for several other purposes.

Here are descriptions of some of Merlin Pro's more commonly used pseudo-opcodes:

DFB—Define a byte of data
DS—Define a data space
ASC—Define an ASCII string
EQU—Equate a symbolic label to a number or a memory location
ORG—Specify origin (starting address) of object code

Some of the more popular assemblers available for the //c are listed in the references at the end of Chapter 2.

Running Assembly-Language Programs

To run an assembly-language program, two steps must take place. The first step is obvious: the program must be loaded into memory. This can be done by storing the bytes that make up the programs into the appropriate area of memory by using Applesoft POKE statements or by using the system monitor STORE command. (See Chapter 3.) The easier method, however, is to load it from the binary file on diskette in which it is contained ("BIN" is displayed to the right of a binary file's name when a diskette is CATALOGued) by using the ProDOS BLOAD command. The BLOAD command must be entered while you are in Applesoft and is of the form:

```
BLOAD FILENAME ,Aaddr
```

where "FILENAME" represents the name of the binary program and "addr" represents the memory location at which it is to be loaded, in hexadecimal (if preceded by "\$") or decimal notation. The ",Aaddr" suffix can be omitted if you wish; if it is, then the file will be loaded into memory at the same position it was in when the BSAVE command was used to save it to diskette.

The second step is to actually run the program. This can be done by using the Applesoft CALL command, which is of the form

```
CALL start
```

where "start" represents the decimal starting address of the program. For example, to run a program that begins at location \$300 (768 decimal), you would enter the command CALL 768. The alternative way of starting the program is to use the system monitor's GO command. (See Chapter 3.) This can be done by entering the system monitor from Applesoft using a CALL - 151 command and then, for a program beginning at location \$300, entering the command "300G".

Some of the programs in this book will not operate properly if they are loaded and called in this way (they will be specifically noted). Instead, the ProDOS BRUN command must be used to load and execute them directly from diskette. This command can be entered as follows:

```
BRUN FILENAME
```

where "FILENAME" represents the name of the binary program. When the

BRUN command is used, the program will be loaded into memory at the location from which it was saved to diskette using the ProDOS **BSAVE** command. To save a copy of a binary program that you have already entered into memory to a diskette, enter the command:

```
BSAVE FILENAME , Aaddr , Lnum
```

where “addr” represents the starting address of the program and “num” represents the number of bytes in the program, or the command:

```
BSAVE FILENAME , Aaddr1 , Eaddr2
```

where “addr1” and “addr2” represent the starting and ending addresses, respectively, of the program.

Appendix VI

Periodicals of Interest

The following magazines are excellent sources of information on the Apple //c (and related products). The address given for each magazine is that of the subscription department, which is not necessarily the same as the editorial department.

1. A + , The Independent Guide for Apple Computing

Price: \$24.97/year (\$36.97 in Canada)

Address: P.O. Box 2965
Boulder, Colorado
80321

2. Apple Assembly Line

Price: \$18/year (\$21 in Canada)

Address: P.O. Box 280300
Dallas, Texas
75228

3. Apple Orchard

Price: \$24.00/year (\$30.00 in Canada)

Address: P.O. Box 6502
Cupertino, California
95015

4. Call -A.P.P.L.E.

Price: \$21.00/year (\$36.00 in Canada)

Address: 290 S.W. 43rd
Renton, Washington
98055

5. inCider

Price: \$25.00/year (\$27.97 in Canada)

Address: P.O. Box 911
Farmingdale, New York
11737

6. Nibble, The Reference for Apple Computing

Price: \$26.95/year (\$39.95 in Canada)

Address: 45 Winthrop Street
Concord, Massachusetts
01742

Index

- (dash) 123
- & (ampersand) 95
- /RAM 121
- 6502 2, 11
- 65C02 2, 11–43
 - address space 12, 38–41
 - addressing modes 27–33
 - cycle time 13
 - I/O handling 33–34
 - instruction set 13–20
 - interrupts 34–38
 - registers 21–27
 - stack 12–13, 26–27, 38
 - stack pointer 13, 26–27
 - status flags 23–26
 - zero page 12–13, 38
- 6551 ACIA 299, 303–310
 - interrupt handling 318–324
- 80/40 switch 157, 185
- 80COL switches 177, 188, 192, 194, 211, 212, 222
- 80STORE switches 192, 194, 208, 212, 216, 239–240, 242, 250–251, 316
- ABS 102
- ACIABUF 319, 321
- access code 130
- accumulator 21–22
- addressing modes 27–33
 - absolute 29–30
 - absolute indexed 32
 - absolute indexed indirect 33
 - absolute indirect 33
 - accumulator 30
 - immediate 28–29
 - implied 30
 - indirect indexed 31
 - relative 32
 - zero-page indexed indirect 30–31
 - zero-page indirect 31
- AKD 160–161, 168–173
- ALTCHARSET switches 194–196
- ALTZP switches 234–235, 236–239
- alternative character set 194–196
- ampersand command 95
- animation 220–221
- any-key-down switch (see “AKD”)
- APPEND 125
- Apple I 2
- Apple II
 - announcement 3
 - clones 5
 - patent 1
- Apple II Plus 3, 4
- Apple //c
 - announcement 6
 - back panel 7–8
- Apple //e 5
- Apple /// 4–5
- Applesoft 3, 4, 67–113
 - linking to assembly language 94–97
 - memory map 68–72
 - source code 68
 - tokenization 72–77
 - variables 72
- argument register (ARG) 97, 104
- arithmetic
 - binary 24, 25–26
 - decimal 24, 25–26
- Arkley, John 4
- array variables 83–85
- ARYTAB 71, 72, 78, 82–83
- ASCII codes 141–145
 - negative and positive 141
- assembler
 - formats 49–51
- ATN 102
- Auricchio, Rick 4, 5
- auto-repeat 168–173
- AUXMOVE 241–244, 250
- bank-switched RAM 41, 230–236
- bank switching 229
- BASCALC 63, 190
- BASIC.SYSTEM 116
 - commands 121–126

- BASL 190
- Baudot code 141
- Baum, Allen 2
- Beernink, Ernie 6–7
- BLOAD command 51, 123
- blocks 116
- break instruction (BRK) 34, 37–38
- BRK (see “break instruction”)
- Broedner, Walt 5
- BRUN 123, 156
- BSAVE 123
- BYE 125
- C3COUT1 199–200, 203–205
- C3KEYIN 145, 148, 152
- CALL 95
- CAT 122
- CATALOG 122
- CH 146, 201
- CHAIN 125
- character input subroutines 145–150
- character output subroutines 199–201
- CHARGET 91–93, 97, 98
- CHARGOT 98
- CHKCOM 102, 105, 108, 110
- CLAMP MOUSE 279
- CLEAR MOUSE 279, 280
- CLOSE 125
- CLREOL 63
- CLREOP 63
- CLRSCR 215
- CLRTOP 215
- co-resident programs 245–253
- COLD 103
- COLOR 214
- COLOR = 213
- colors
 - high-resolution 219–220, 223
 - low-resolution 209, 212–213
- communications port 313–314
- CONINT 99, 110
- COS 102
- COUT 65, 199, 203–205
- COUT1 65, 185, 199–200, 203–205
- CR 63
- CREATE 122
- CSW 153, 199–200, 205–206
- CURSOR 148, 153
- CV 146, 201
- cycle time 15, 258
- data bit 301–302
- data format 301–303
- DELETE 123
- DHIRES switches 208, 211, 216, 222
- Disk II 3–4, 115
- directories 119–121
 - format of 127–130
- directory file entry 129
- directory header 128
- disk drive 115
 - booting 115
 - external 115
- diskettes
 - formatting 116
- display attributes 194–196
- DISVBL 285
- DISXY 285
- DOCTL 201
- DOS 3.3 4
- DRAW 224, 226
- Dvorak keyboard 157
- EBCDIC codes 141
- effective address 28
- ENBXY 285
- ENVBL 285
- ERRFLAG 102, 103
- ERROR 102
- escape sequences 148–150
- ESCRDKEY 145, 150
- EXEC 124
- EXP 102
- EXTINT2 323
- FADD 100
- FACLO 99, 103
- FDIV 100
- filenames 118–121
- file types 130
- flags (see “status flags”)
- flash video 194–196, 203–205
- floating point accumulator (FAC) 96–97, 99, 100, 101, 104, 110
- FLUSH 125

- FMULT 100
- FNDLIN 98
- FOUT 101
- FRE 125
- FRESPC 100, 101, 103
- FRETOP 71, 72, 79, 81–82, 100, 101
- FRMEVL 99, 101
- FRMNUM 99, 108
- FSUB 100
- functions 82
- FX0EDGE 285
- FY0EDGE 285
- game controller 265, 288–293
- GARBAGE 101
- garbage collection 82, 101, 125
- GBASCALC 215
- GBASL 214
- GETADR 100, 110
- GETARYPT 98
- GETBUFR 117–118
- GETBYT 99
- GETLN 64, 145, 150–152
- GETSPACE 100
- GIVAYF 99, 110
- GOTKEY 148, 154
- GR 213
- H2 214
- HCOLOR = 224
- HGR 224, 226
- HGR2 224, 226
- HHORIZ 225
- high-resolution graphics mode
 - 214–227
 - animation 220–221
 - commands 223–225
 - double-width 221–223
 - double-width colors 223
 - double-width memory mapping 222–223
 - single-width colors 219–220
 - single-width memory mapping 217–218
 - turning on double-width
 - turning on single-width 215–217
- HIMEM: 71, 72
 - and ProDOS 117–118
- HIRES switches 208, 210, 222, 239–240
- HLIN (high-res) 226
- HLIN (low-res) 213
- HLINE 215
- HMASK 225
- Holt, Rod 1
- HOME 63
- HOMEMOUSE 280
- HPAG 225
- HPlot 224, 226
- Huston, J.R. 4, 6–7
- HVERT 225
- IN# 125, 152–153, 155–157
- index registers 22–23
- INITMOUSE 280
- input buffer 39, 69
- Input/Output memory 41–42
- input link 147, 152–153
 - effect of ProDOS 154–157
 - ProDOS link 156
- instruction pointer (see “program counter”)
- instructions (65C02) 13–20
- Integer BASIC 2, 3
- integer numbers 85–86
- interrupt requests (IRQ) 34, 36–37
- interrupts 24–25, 34–38
 - external 323–324
 - mouse 267–268, 280
 - serial interface 318–324
- inverse video 194–196, 203–205
- INVFLG 204
- IOUDIS switches 208, 211, 216, 222, 285
- IRQ (see “interrupt requests”)
- Jobs, Stephen 1–2
- KBD 160–161, 173, 321–323
- KBDSTRB 160–161, 168–173, 323
- key block 127
- keyboard 157–181
 - 80/40 switch 157
 - auto-repeat 168–173
 - I/O locations 160–161
 - interrupts 321–323
 - keyboard switch 157

- reset key 173–181
- strobe 160–161
- keyboard input 148–150
 - modifying 153–154, 164–168
- KEYIN 64, 145, 148, 152–154, 185
- keywords 74–77
- KSW 147, 152–153, 164
- Language Card 4
- line input subroutines 150–152
- LINGET 102
- links 60, 147
- LINNUM 98, 100, 102, 103, 110
- LINPRT 100
- Lisa 5
- LOAD 124
- LOCK 123, 130
- LOG 102
- LOMEM: 71
- low-resolution graphics mode 207–214
 - commands 213–214
 - double-width 210–213
 - double-width colors 212–213
 - double-width memory mapping 212
 - single-width colors 209
 - single-width memory mapping 209
 - turning on double-width 210–211
 - turning on single-width 208–209
- LOWTR 98, 103
- machine language interface 132–134
- Macintosh 6
- Markkula, Mike 2, 6
- MASK 214
- memory map
 - auxiliary RAM memory 41, 236–245
 - bank-switched RAM 41
 - Input/Output memory 41–42
 - main RAM memory 38–41
 - ROM memory 42
- memory-mapped I/O 33–34, 183–184
- MEMSIZ 71, 72, 81
- Merlin Pro assembler 9–10, 49
- Microsoft 3, 67
- MIXED switches 208–209, 210, 216, 222
- MLI (see “machine language interface”)
- MON 45–46
- MONZ 45–46, 65
- mouse 265–288
 - Applesoft control 269–273
 - assembly language control 273–280
 - comparison with //e mouse 275–278
 - how it works 266–267
 - joystick emulation 280–281
 - operating modes 267–268
 - screen hole usage 274
- MouseText 183, 196–199
- MOUSEXINT 288
- MOUSEYINT 288
- MOUSTAT 268, 278, 279, 280
- MOUX1 288
- MOUY1 288
- MOVESTR 100, 101
- musical notes 255–259
- NEWESC 148
- NMI (see “non-maskable interrupts”)
- non-maskable interrupts (NMI) 34
- OFFBSR1 232
- OFFBSR2 232
- OLDRST 45–46
- opcode 27–28
- OPEN 124
- OPEN-APPLE key 159, 173–174, 295, 322
- OURCH 201
- OURCV 201
- output link 153, 199–200, 205–207
 - effect of ProDOS 206–207
- PAGE2 switches 192–194, 208, 212, 216, 239–240
- page 3 vectors 118

- page of memory (definition) 12
- parity bit 302
- Pascal 4, 230
- pathnames 118–121
- PDL inputs 289–293
- photoresistor 292–293
- pixels 214, 216
- PLOT (high-res) 215
- PLOT (low-res) 213
- port assignments 9
- port selection 60
- POSITION 125
- POSMOUSE 279
- PR# 125, 205, 207
- PRBYTE 64
- PREFIX 120–121, 123
- PRGEND 71
- PREAD 63, 285, 290
- prefix 120–121
- primary character set 194–196
- printer port 311–313
- PRINTYX 62, 63
- PRNTFAC 100, 108
- processor status register 23
- ProDOS 115–140
 - announcement 6
 - memory map 116–118
- program counter 27
- prompt symbols 151
- PTRIG 288, 289–290, 292
- PTRGET 98, 104–108
- push buttons 293–296
- PWREDUP 174–175
- Quinn, Peter 5
- RAMRD switches 237, 239–240, 250–251
- RAMWRT switches 237, 239–240, 250–251
- RD63 288
- RD80SW 185
- RDBANK2 232–234
- RDCHAR 64, 145, 150
- RDKEY 64, 145–150, 152
- RDLGRAM 232–234
- RDVBLMSK 288
- RDX0EDGE 288
- RDY0EDGE 288
- RDXYMSK 288
- READ 125
- READBSR1 232
- READBSR2 232
- READMOUSE 268, 278, 279
- real numbers 87–89
- registers 21–27
 - accumulator 21–22
 - index (X and Y) 22–23
 - processor status 23
 - program counter 27
- RENAME 123
- RESET 174
- reset interrupt 34, 36, 173–181
 - trapping reset 174–181
- RESTORE 126
- ROM memory 42
- ROT 225
- ROT = 224
- RS-232-C standard 300, 304–305
- RSTXY 288
- RUN 124
- RX0EDGE 285
- RY0EDGE 285
- sapling file 131–132
- SAVE 124
- SCALE 225
- SCALE = 224
- SCRN (high-res) 215
- SCRN (low-res) 213
- Sculley, John 6
- seedling file 131
- serial interface 299–303
 - configuration 310
- SERVEMOUSE 279, 280
- SETCOL 215
- SETHCOL 226
- SETMOUSE 278
- Shepardson, Bob 4
- Sholes keyboard 157
- SHOWCUR 148, 154
- simple variables 78–83
- SIN 102
- SOFTTEV 174–175
- SOLID-APPLE key 159, 295

- speaker 255–263
 - I/O location 255
- SQR 102
- stack 12–13, 26–27, 38, 69
- stack pointer 13, 26–27
- start bit 301
- status flags 23–26
 - break 25
 - carry 23–24
 - decimal mode 25
 - interrupt disable 24–25
 - negative 26
 - overflow 25–26
 - zero 24
- Stearns, Bryan 5
- stop bit 302
- STORCH 200
- STORE 126
- STORY 154
- STREND 71, 72, 78, 82, 85
- STROUT 101
- STRPRT 101
- “Sweet 16” 2
- system monitor 45–65
 - ADD command 57
 - BASIC command 57
 - command syntax 46, 61
 - CONTINUE BASIC command 57–58
 - DISPLAY command 46, 48–49
 - entry points 45–46
 - EXAMINE command 53–54, 62
 - GO command 54–55, 62
 - INVERSE command 57
 - KEYBOARD command 59–61
 - LIST command 55–56
 - MOVE command 51–53
 - NORMAL command 57
 - PRINTER command 59–61
 - STORE command 49–51
 - subroutines 61–65
 - SUBTRACT command 57
 - USER command 58–59
 - VERIFY command 53
 - TAN 102
 - terminal mode 314–316
 - text mode 184–207
 - memory mapping 188–194
 - turning it on 185–188
 - TEXT switches 188, 208, 210, 216, 222
 - thermistor 292
 - tree file 131–132
 - TRKEY 322
 - TWKEY 322
 - two’s complement 85–86
 - TXTPTR 90–94, 98, 99, 102, 103, 104–108
 - TXTTAB 70, 73, 194
 - TYPHED 322, 323
 - UNLOCK 123, 130
 - UPDATE 154
 - USR 96–97, 104
 - V2 214
 - variables 72, 77–85
 - evaluating 108
 - locating 104–108
 - VARPNT 98, 103, 104–108
 - VARTAB 71, 72, 78
 - VBL interrupt 221, 267, 268, 278
 - VBLINT 288
 - VERIFY 123
 - vertical blanking interrupt (see “VBL interrupt”)
 - VFACTV 176–177
 - video RAM
 - auxiliary memory 191–192
 - high-resolution pages 40–41, 72, 215–218, 222–223, 236
 - low-resolution pages 208–209, 212
 - text pages 40, 69, 189–194, 236
 - VIDOUT 200
 - VIDOUT1 201
 - VIDWAIT 200
 - VisiCalc 4
 - VLIN 213
 - VLINE 215
 - VMODE 148, 150
 - volume bit map 126
 - VTAB 63
 - WAIT 64

WARM 103
Wigginton, Randy 3, 4
Williams, Rich 6–7
windows 202–203
WNDBTM 203
WNDLFT 203
WNDTOP 203
WNDWIDTH 203
Wozniak, Stephen 1–2

WRITE 125
WRITEBSR1 232
WRITEBSR2 232
XDRAW 224, 226
XFER 244–245
XRDSER 320
zero page 12–13, 38, 69
 free space chart 41

RELATED RESOURCES SHELF

Apple IIc:

An Introduction to Applesoft BASIC

Lois Graff, Larry Joel Goldstein

Designed for both novices and experienced users, this unique guide leads you through the fundamentals of BASIC while familiarizing you with the many capabilities of your Apple IIc.

☐ 1984/371pp/paper/D2913-4/\$16.95

Apple IIc User Guide *Gary Phillips, Donald Scellato*

Presents a clear account of the programming capabilities and applications for the Apple IIc. Friendly, step-by-step instructions are delivered throughout.

☐ 1984/369pp/paper/D3073-6/\$14.95

☐ Book-Diskette/1984/D2921-7/\$34.95

☐ Diskette/1984/D293X-3/\$20.00

Brain Games for Kids and Adults Using the Apple II/IIe/IIc

John W. Stephenson, Ph.D., Robert L. Randell, Ph.D.

(Softsync)

This remarkable book packs an enormous amount of information in just a few lines of code to give you more than games alone. It's a learning tool! Guess at the solutions or work them out logically.

☐ 1984/231pp/paper/D3626-1/\$13.95

☐ Diskette/1984/D3669-1/\$20.00

TO ORDER, simply clip or photocopy this entire page, check off your selection, and complete the coupon below. Enclose a check or money order for the stated amount. (Please add \$2.00 postage and handling per book plus local sales tax.)

Mail to: **Prentice-Hall, Inc., P.O. Box 462, West Nyack, NY 10994**

Name _____

Address _____

City/State/Zip _____

Charge my credit card instead: ☐ MasterCard ☐ Visa

Account# _____ Expiration Date _____

Signature _____

Dept. Y

Y0510-BB(5)

Prices subject to change without notice.

"What can I say that is not superlative. . .this book has the potential of being the definitive technical guide on the Apple //c!"

"For a technical orientation, no other book covers the variety of topics this does!"

INSIDE THE APPLE //c

Gary B. Little

Now—a comprehensive look at the advanced features and capabilities of the Apple //c! This book presents an insider's view of the 65C02 microprocessor that controls the //c, ProDOS, the //c system monitor commands; how the //c handles character input and output; memory management techniques; how to control the speaker, mouse, and game controller; how to use the //c's two built-in serial ports for communications with printers and modems, plus much more.

Here are just a few of the many interesting programming examples you'll find in this book:

- How to speed up the auto-repeat rate of the cursor (using software techniques only)
- How to run two Applesoft programs concurrently (one in main memory and the other in auxiliary memory)
- How to read mouse input using 65C02 interrupt techniques
- How to read and write specific blocks on the ProDOS-formatted diskette
- How to use the keyboard "type-ahead" feature

CONTENTS

An Introduction to Apple and the Apple //c • The 65C02 Microprocessor • The System Monitor • Applesoft BASIC • The ProDOS Disk Operating System • Character Input and the Keyboard • Character and Graphic Output and Video Display Modes • Memory Management • The Speaker • Mouse and Game Controller Input • The Serial Interface Ports • Appendix I: American national Standard Code for Information Interchange (ASCII) Character Codes • Appendix II: 65C02 Instruction Set and Cycle Times • Appendix III: Apple //c Soft Switch, Status, and I/O Port Locations • Appendix IV: Apple //c Page3 Vectors • Appendix V: For Beginners Only • Appendix VI: Periodicals of Interest • Index



ISBN 0-89303-564-5